EL MENSAJERO DE PEKIN

Pere Fonolleda
Ferran Font

Setembre 2009

Índex de continguts

1.Introducció....13
1.1.Motivació....13
1.2.Objectius del Projecte....14
1.3.Temporització del Projecte....16
1.4.Organització del document....17
1.5.Separació de tasques....18
1.5.1.Pere Fonolleda....19
1.5.1.1.Implementació de l'animació per óssos....19
1.5.1.2.Implementació de la interfície gràfica....19
1.5.1.3.Implementació del multijugador....20
1.5.2.Ferran Font....20
1.5.2.1.Implementació de la física....20
1.5.2.2.Implementació de la música....20
1.5.2.3.Implementació de la intel·ligència artificial....20
2.Estudi previ....22
2.1.Introducció als motors de videojocs....22
2.2.Diferències entre un motor gràfic i un motor de videojocs....24
2.3.Cerca de motors de videojocs....24
2.3.1.Ogre3D....25
2.3.2.Crystal Space....26
2.3.3.Irrlicht....27
2.3.4.jMonkey Engine....28
2.3.5.Yake....29
2.3.6.OGE....30
2.3.7.Raydium....31
2.3.8.OpenSceneGraph....32
2.3.9.Delta3D....33
2.4.Elecció del jMonkey Engine....33
2.5.Descripció d'alguns conceptes teòrics d'informàtica gràfica relacionada amb els videojocs....34

2.5.1.Vector....34
2.5.2.Bone (Ós)....35
2.5.3.Caixa envolupant (BoundingBox)....35
2.5.4.Skybox....36
3.Eines utilitzades....38
3.1.Sistema operatiu....38
3.2.Llibreries utilitzades....38
3.2.1.jMonkey Engine 38
3.2.1.1.Introducció al jMonkey Engine 38
3.2.1.2.Funcionament del jMonkey Engine 39
3.2.2.Monkey World 3D....44
3.2.3.jME 2 Physics 2....45
3.2.3.1.Funcionament....46
3.2.3.2.Classes principals....47
3.2.4.GBUI 49
3.2.4.1.Funcionament 49
3.2.4.2.Classes Principals....50
3.2.5.OpenGL....52
3.2.6.OpenAL....53
3.2.7.LWJGL....54
3.3.Programes utilitzats....55
3.3.1.Eclipse IDE....55
3.3.2.Blender....56
3.3.3.Audacity....57
3.3.4.Gimp....57
3.3.5.OpenOffice.org....58
3.3.6.Dia....59
4.Descripció del videojoc....61
4.1.Descripció general....61
4.2.Ambientació i argument....61
4.3.Funcionament del joc....61
4.3.1.Modalitats....63

4.3.2.Atributs....64
4.3.3.Moviments....64
4.3.3.1.Moviment dels personatges....64
4.3.3.2.Llançament de paquets....64
4.3.4.Adversitats....65
4.3.4.1.Rebre l'impacte d'un paquet....65
4.3.4.2.Estar retingut per un PNJ....65
4.4.Opcions del videojoc fora de la partida....66
5.Requeriments del Sistema....67
5.1.Requeriments Funcionals....67
5.1.1.Requeriments generals del sistema....67
5.1.2.Identificació dels actors....68
5.1.3.Diagrames de casos d'ús generals....69
5.1.3.1.Diagrama de casos d'ús: Menú principal....69
5.1.3.2.Diagrama de casos d'ús: Partida....70
5.1.4.Detalls dels requeriments del sistema....71
5.1.4.1.Cas d'us: Mostrar crèdits....71
5.1.4.2.Cas d'ús: Mostrar màximes puntuacions....71
5.1.4.3.Cas d'ús: Moure's (sense paquets)....72
5.1.4.4.Cas d'ús: Recollir paquets (jugador sense paquets)....72
5.1.4.5.Cas d'ús: Moure's (jugador amb paquets)....73
5.2.Requeriments no funcionals....73
6.Anàlisi del sistema....75
6.1.Casos d'ús refinats....75
6.1.1.Casos d'ús del Menú principal....76
6.1.1.1.Cas d'ús: Inicia partida d'un jugador....76
6.1.1.1.1.Fitxa del cas d'ús....76
6.1.1.2.Cas d'ús: Inicia partida multijugador....77
6.1.1.2.1.Fitxa del cas d'ús....77
6.1.1.3.Cas d'ús: Accedir a les opcions....78
6.1.1.3.1.Fitxa del cas d'ús....78
6.1.2.Casos d'ús de partida....79

6.1.2.1.1.Fitxa del cas d'ús: Llençar paquet a un altre jugador.....79
6.1.2.1.2.Fitxa del cas d'ús: Llençar paquet a una bústia.....80
6.1.2.1.3.Fitxa del cas d'ús: Llençar paquet a un personatge no jugador. 80
6.1.2.1.4.Fitxa del cas d'ús: Llençar paquet a l'escenari.....81
6.2.Diagrames d'activitat.....81
6.2.1.Diagrama d'activitat: Iniciar partida.....82
6.2.2.Diagrama d'activitat: Accedir i modificar opcions.....84
6.2.3.Diagrama d'activitat: Veure crèdits i màximes puntuacions.....86
6.2.4.Diagrama d'activitat: Partida.....87
7.Disseny del sistema.....89
7.1.Diagrames de classes.....89
7.1.1.Descripció de les classes.....93
7.1.1.1.Classes parcials del diagrama de classes relacionat amb les pantalles principals.....93
7.1.1.1.1.ElMensajeroDePekin.....93
7.1.1.1.2.StandardGame.....94
7.1.1.1.3.ImplCallBack.....95
7.1.1.1.4.GestorMusica.....96
7.1.1.1.5.GestorPantalla.....98
7.1.1.1.6.GestorFitxers.....98
7.1.1.1.7.PhysicsGameState.....99
7.1.1.1.8.GbuiGameState.....99
7.1.1.1.9.MenuPrincipal.....101
7.1.1.1.10.MenuIniciarPartidaIndividual.....101
7.1.1.1.11.MenuIniciarPartidaMultijugador.....103
7.1.1.1.12.MenuConfiguracio.....103
7.1.1.1.13.Credits.....104
7.1.1.1.14.Puntuacions.....104
7.1.1.1.15.Partida.....105
7.1.1.1.16.MenuDePausa.....107
7.1.1.1.17.GuardarPuntuacions.....107
7.1.1.2.Classes parcials del diagrama de classes relacionat amb una partida

.....107
7.1.1.2.1.GestorModels.....108
7.1.1.2.2.SimpleSplatManager.....108
7.1.1.2.3.Escenari.....108
7.1.1.2.4.Bustia.....110
7.1.1.2.5.FurgonetaDeCorreos.....110
7.1.1.2.6.Personatge.....111
7.1.1.2.7.PersonatgeNJ.....112
7.1.1.2.8.Paquet.....112
7.1.1.2.9.RegistrePuntuacio.....113
7.1.1.3.Classes parcials del diagrama de classes relacionat amb els controladors.....114
7.1.1.3.1.InputHandler.....114
7.1.1.3.2.ThirdPersonHandler.....114
7.1.1.3.3.EMdPHandler.....115
7.1.1.3.4.KeyInputAction.....115
7.1.1.3.5.EndavantEMdPAction.....116
7.1.1.3.6.EndarreraEMdPAction.....116
7.1.1.3.7.DretaEMdPAction.....116
7.1.1.3.8.EsquerraEMdPAction.....117
7.1.1.3.9.DisparарEMdPAction.....117
7.1.1.3.10.PausarEMdPAction.....117
7.2.Disseny de la persistència.....117
7.2.1.1.Emmagatzemament de dades.....118
7.2.1.1.1.Fitxer de configuració: config.emdp.....118
7.2.1.1.2.Fitxer de dades d'escenaris: llistatEscenaris.emdp.....119
7.2.1.1.3.Fitxer de dades dels personatges: llistatPersonatges.emdp.…120
7.2.1.1.4.Fitxers de registre de puntuacions: minimTemps.emdp i maximPaquets.emdp.....121
8.Implementació.....122
8.1.La classe ElMensajeroDePekin.....122
8.2.Els estats.....123

8.2.1.GbuiGameState....123
8.2.2.Estats per a iniciar partides....124
8.2.2.1.Iniciar i posar un títol....125
8.2.2.2.Imatges i fletxes per a navegar-hi....125
8.2.2.3.Crear els botons del mode....128
8.2.2.4.Botons per a iniciar partida o retornar....129
8.3.Gestors....130
8.3.1.El gestor de pantalla....130
8.3.2.Càrrega de models i textures: el gestor de models....132
8.3.3.Lectura i escriptura a fitxers: GestorFitxers....134
8.4.Les classes Bàsiques....134
8.4.1.Personatge....134
8.4.2.Classe PersonatgeNJ....138
8.4.3.Classe FurgonetaDeCorreos....140
8.4.4.Escenari....140
8.5.La classe partida....141
8.6.Física....143
8.6.1. Col·lisió d'un paquet contra un personatge o un PNJ....144
8.6.2.Col·lisió d'un paquet contra una bústia del mateix color....145
8.6.3.Col·lisió d'un paquet contra una bústia de colors diferents....146
8.6.4.Col·lisió d'un personatge amb un personatgeNJ....146
8.6.5.Col·lisió d'un personatge amb una furgoneta....147
8.7.Control d'entrades....148
8.7.1.Classe EMdPHandler....148
8.7.2.Accions....149
8.7.2.1.Accions de moviments....150
8.7.2.2.Acció de disparar....151
8.7.2.3.Acció del Menú de Pausa....151
8.8.Implementació de models i optimització dels recursos....152
8.8.1.Número de polígons....152
8.8.2.Textures....153
9.Resultats....156

9.1.Resultats de l'apartat comú....156
9.2.Resultats dels apartats d'en Pere Fonolleda....156
9.2.1.Animació per óssos....157
9.2.2.Disseny de la interfície....160
9.2.3.Multijugador....162
9.3.Resultats dels apartats d'en Ferran Font....163
9.3.1.Disseny de la física....163
9.3.2.Disseny de la Intel·ligència Artificial....164
9.3.3.Disseny de la Música....164
10.Conclusions....165
10.1.Conclusions personals....165
11.Treball futur....166
12.Agraïments....167
13.Bibliografia....168

Índex de figures

Figura 1: Imatge del joc quake III....13
Figura 2: Imatge del videojoc Bomberman Online....15
Figura 3: Diagrama de gantt de la temporització del projecte....16
Figura 4: Esquema de les capes d'un motor de videojocs....23
Figura 5: Imatge del joc comercial Ankh....25
Figura 6: Imatge del MMORPG PlaneShift creat amb CrystalSpace....26
Figura 7: Render de demostració d'efectes de reflex i partícules del motor....27
Figura 8: Imatge del joc Spirits....29
Figura 9: Captura de pantalla del plugin OGEd....31
Figura 10: Imatge del joc NewSkyDriver creat amb Raydium....32
Figura 11: Representació d'un vector en el pla....34
Figura 12: Exemple d'una estructura d'óssos amb deformació de malla....35
Figura 13: Imatge representativa de BoundigBox en cub i esfera....36
Figura 14: Imatge que ens mostra com funcionen els Skyboxes....37
Figura 15: Diagrama de les classes bàsiques que hereten de AbstractGame....40
Figura 16: Diagrama de classes bàsic que hereta de GameState....42
Figura 17: Diagrama bàsic de les classes principals dels elements de l'escena....43
Figura 18: Exemple d'un possible arbre de nodes....44
Figura 19: Captura de pantalla del Monkey World 3D....45
Figura 20: Vista del debug mode del motor de física Physics....46
Figura 21: Esquema d'un graf d'escena amb nodes físics....47
Figura 22: Diagrama de les classes més importants del Physics....48
Figura 23: Imatge d'una captura de pantalla de joc Bang! Howdy....49
Figura 24: Exemple d'herencia entre elements de GBUI....50
Figura 25: Arbre d'herència parcial de BComponent....51
Figura 26: Part de l'abre d'herència de BComponet que mostra les finestres....52
Figura 27: Exemple d'una renderització amb el programa blender d'un pla tal i com volem que quedi col·locada la càmera....63
Figura 28: Identificació dels actors....68
Figura 29: Diagrama de cas d'ús: Menú principal....69

Figura 30: Diagrama de casos d'ús: Partida.....70
Figura 31: Cas d'ús: Inicia partida d'un jugador.....76
Figura 32: Cas d'ús: Inicia partida multijugador.....77
Figura 33: Cas d'ús: Accedir a les opcions.....78
Figura 34: Casos d'ús de partida.....79
Figura 35: Diagrama de classes parcial en el qual es veuen les pantalles principals90
Figura 36: Diagrama de classes parcial en el qual es veuen les classes relacionades amb una partida.....91
Figura 37: Diagrama de classes parcial en el qual es veuen les classes relacionades amb els controladors.....92
Figura 38: Exemple de la configuració d'un escenari.....120
Figura 39: variable estàtica _joc de la classe StandardGame.....122
Figura 40: Inici del “main” d'El Mensajero De Pekín.....122
Figura 41: Constructor de la classe GbuiGameState.....123
Figura 42: Mètodes activate i deactivate de la classe GbuiGameState.....124
Figura 43: Fragment del mètode MenuIniciarPartidaUnJugador.....125
Figura 44: Imatge de com ha de quedar el selector de personatges i escenaris.....126
Figura 45: Codi per a crear els contenidors per a imatges.....126
Figura 46: Listener que conté el botó d'anar endarrere en el selector d'escenari....127
Figura 47: Listener que conté el botó d'anar endavant en el selector d'escenari.....128
Figura 48: Codi d'un boto toggle que al activar-se desactiva l'altre mode.....129
Figura 49: Codi per als botons de tornar enrere i jugar.....130
Figura 50: Mètode gestioPantalla.....131
Figura 51: Mètode inicialitzaPantalles.....131
Figura 52: Mètode canvia.....132
Figura 53: Mètode iniciaPartida per a mode multijugador.....132
Figura 54: Implementació del mètode getJMEXML.....133
Figura 55: Creació de la textura i afegiment al personatge.....133
Figura 56: Càrrega d'un model en 3DS.....134
Figura 57: Imatge de la combinació de dos Bounding Box per a gestionar les col·lisions d'un personatge.....135
Figura 58: Nodes de física d'un model.....136

Figura 59: Codi per a crear la caixa envolupant dels peus del personatge.....136
Figura 60: Codi per a crear i rotar la càpsula que envolta al personatge.....137
Figura 61: Codi per a carregar l'animació d'esperar.....137
Figura 62: Mètode carregar.....138
Figura 63: Fragment del mètode buscarAlVoltant.....139
Figura 64: Mètode hiHaJugadorDavant.....140
Figura 65: Mètode actualitzar de la classe FurgonetaDeCorreos.....140
Figura 66: Codi per a crear caixes al voltant dels objectes o_invisible.....141
Figura 67: Part del codi per a crear els personatges a l'inici d'una partida.....142
Figura 68: Creació dels controls d'entrada.....143
Figura 69: Divisió global de les col·lisions segons provinguin d'un paquet o d'un personatge.....144
Figura 70: Codi que s'executa quan xoquen un paquet i un personatge.....145
Figura 71: Col·lisió d'un paquet contra una bústia del mateix color.....145
Figura 72: Codi que s'executa al xocar un paquet contra una bústia de diferent color 146
Figura 73: Comprovació de que els personatges no estiguin bloquejats.....147
Figura 74: Col·lisió de furgoneta amb personatge.....147
Figura 75: Actualització de les propietats i de les tecles de control.....148
Figura 76: Inicialització de l'acció de disparar.....148
Figura 77: Mètode update de la classe EMdPHandler.....149
Figura 78: Mètode performAction.....150
Figura 79: Codi que s'executa al disparar un paquet.....151
Figura 80: Mètode que s'executa al pausar una partida.....152
Figura 81: Mostra de les cases i tanques sense la part posterior.....153
Figura 82: Detall de l'escenari amb textures ja ombrejades.....154
Figura 83: Imatge en detall de ombres simulant relleu.....155
Figura 84: Caputra de pantalla de l'escenari carregat.....156
Figura 85: Unió del model amb l'esquelet.....157
Figura 86: Animació del personatge amb el programa Blender.....158
Figura 87: Models carregats i animats.....159
Figura 88: Menú principal.....160

Figura 89: Menú d'iniciar una partida en mode multijugador.....161
Figura 90: Imatge de la partida, on es veuen els icones indicadors.....162
Figura 91: Seqüència de joc en multijugador.....162
Figura 92: Seqüència que representen la col·lisió de un Personatge contra els elements de l'escenari.....163
Figura 93: Representació de com rebota un paquet per l'escenari.....164
Figura 94: Seqüencia d'entrada i sortida d'una furgoneta a l'escenari.....164

1. Introducció

La industria dels videojocs, avui en dia, és molt important en el món de l'oci, arribant a superar a indústries més clàssiques com la música o el cinema, fet pel qual existeixen molts tipus de videojocs i és difícil innovar a nivell de jugabilitat, sense entrar en el món del hardware, o projectes grans de companyies importants. En quan a software lliure, aquest volum es redueix bastant, i encara es limita més quan entrem al terreny dels jocs multiplataforma/multijugador, tot i que un bon exemple de joc d'aquest estil, comercial i molt conegut, és la saga Quake, que podem veure a la Figura 1.

Figura 1: Imatge del joc quake III

1.1. Motivació

La nostra motivació per a treballar en aquest projecte és la de conèixer el mon del desenvolupament de videojocs.

És una branca de la informàtica que ens interessa, i en la qual segurament vulguem seguir més endavant en la nostra vida professional. Pensem que el projecte final de carrera és l'escenari perfecte per a entrar-hi, ja que podem treballar aquest tema amb un tutor més experimentat que ens supervisi, ens ajudi, i en el nostre cas, ens

transmeti entusiasme, factors que en un projecte d'empresa pròpia no hauríem trobat.

Una motivació addicional és el fet que hem comentat anteriorment, dels pocs videojocs a nivell de software lliure, camp en el qual pensem que podem aportar el nostre petit gra de sorra per, de mica en mica, fer pujar la comunitat dedicada a aquesta branca del programari, i que acabi sent tant important com les comunitats que ja hi ha de software professional (alguns programes dels quals hem fet servir per a la realització d'aquest projecte).

1.2. Objectius del Projecte

El nostre objectiu és el desenvolupament d'un videojoc (senzill) des de un idea inicial fins a un producte acabat i funcional.

Això implica fer tota la feina que comporta el desenvolupament d'un videojoc, des de el disseny del joc (la historia, els nivells, els personatges, etc.) com a concepte, passant per la programació, el disseny gràfic, i finalment el poliment del producte per a que sigui un joc complet.

La nostre primera idea va ser la de crear un videojoc amb una línia argumental més seria, un desenvolupament seguint un format *RPG* (*Role-playing game*, o joc de rol, basat en unes característiques del personatge i una lluita per torns) o *Action-RPG* (similar a l'anterior però amb l'acció en temps real). Finalment, vam descartar aquesta idea ja que un joc d'aquest estil hagués dut molt més temps de guió i *Game Design*, sobretot tenint en compte que el concepte de personatge adquiria unes dimensions molt importants, a causa del nombre de característiques i interaccions amb l'entorn. A més, el guió d'aquests jocs és llarg i complex, així que, apart de ser un tema en el qual nosaltres no treballem (el guió), l'haguéssim deixat incomplet.

Per aquest motiu, finalment s'ha optat per el desenvolupament d'un joc molt més senzill, de partides ràpides (com el videojoc *Bomberman* que podem veure a la Figura 2), i sense un guió molt elaborat. Així ens podem centrar en el que realment interessa com a informàtics, que és el desenvolupament de la seva programació, treballar amb les eines necessàries per a desenvolupar-ho i entrar a conèixer aquest mon tant interessant que no hem pogut tocar durant la carrera.

Figura 2: Imatge del videojoc Bomberman Online

Així doncs, el videojoc que finalment desenvoluparem s'anomenarà "El Mensajero de Pekín", i el l'objectiu com a jugadors serà aconseguir repartir uns paquets a una sèrie de bústies seguint una lògica de colors (cada paquet haurà de ser introduït a la bústia del mateix color per a sumar punts). Ho podrem fer tant a mode individual, en el qual haurem de superar uns reptes que ens posi la màquina, com en mode multijugador, de entre dos i quatre jugadors, en el qual apart de repartir els paquets, haurem d'impedir que els rivals reparteixin els seus.

Per tant, a mode de resum, el l'objectiu serà el desenvolupament complet d'un videojoc senzill.

1.3. Temporització del Projecte

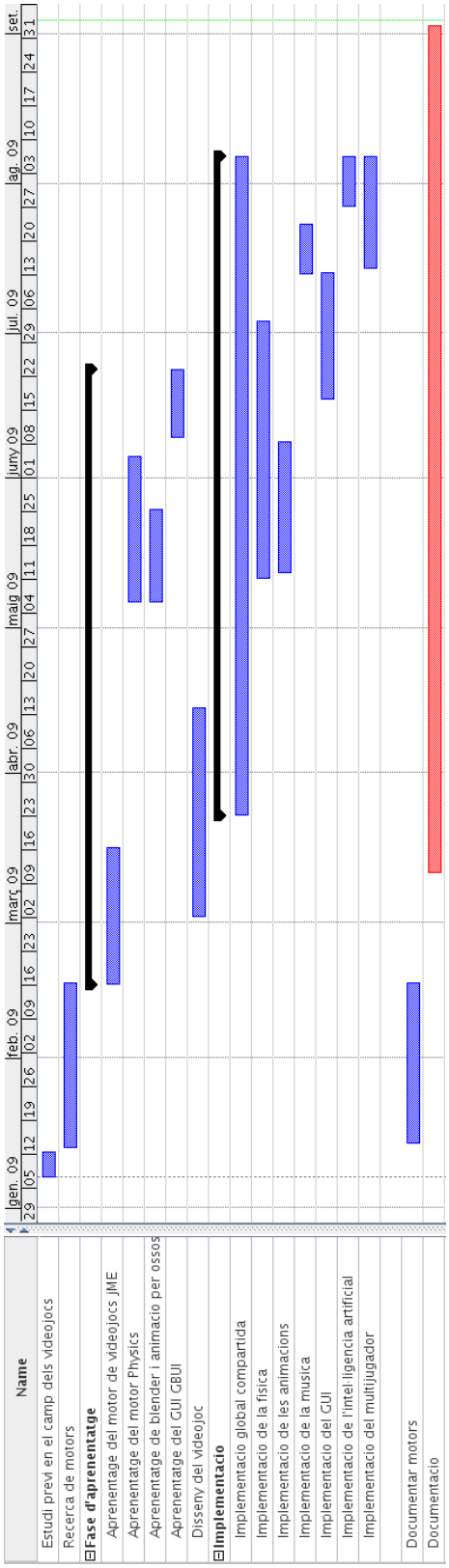


Figura 3: Diagrama de gantt de la temporització del projecte

A la Figura 3 es pot veure la temporització del projecte.

Les tasques comunes i individuals estan explicades a l'apartat: 1.5 Separació de tasques.

1.4. Organització del document

Aquest document s'ha organitzat en nou capítols, que són els següents:

- **Introducció**. En aquest capítol es vol explicar el perquè s'ha desenvolupat aquest projecte, quin son els objectius que proposats, com ha estat organitzat el desenvolupament i com s'han separat les tasques entre els dos integrants, o quines tasques s'han fet comunes.
- **Estudi previ**. En aquest apartat es tractaran les principals tasques desenvolupades durant les primeres etapes de realització del videojoc. Aquí s'inclouen els passos d'estudi i aprenentatge de conceptes que s'han fet servir per al desenvolupament.
- **Eines utilitzades**. Aquesta secció conté un repàs de les eines utilitzades, amb la seva descripció i el perquè s'han fet servir, tant de llibreries i motors com de programari.
- **Descripció del videojoc**. Conegut com a *Game Design*, aquesta secció conté l'apartat a on s'explica el funcionament conceptual del videojoc, que ha de tenir, com ha de funcionar, que farà i no farà el jugador, etc.
- **Requeriments del sistema**. En aquest capítol es definiran els *Requeriments del programari*, els quals recullen, a grans trets, els objectius de l'aplicació juntament amb les funcionalitats que es volen obtenir. Aquest document ens ha de permetre entendre els elements que envoltaran el sistema informàtic que s'intenta construir.
- **Anàlisi del sistema**. Aquest apartat pretén que s'obtingui una comprensió precisa de les necessitats del sistema. És a dir, s'encarrega de la investigació del problema a resoldre (que), però no s'interessa en trobar-ne una solució (com). Fent servir Enginyeria del Software, en aquesta secció traduirem els requeriments esmentats en capítols anteriors a un llenguatge més formal.

- **Disseny del sistema**. Aquí el que procurarem és, partint del capítol anterior, anar més enllà en quant a especificació es refereix, i fer un esquema d'implementació del sistema mitjançant diverses eines de programació orientada a objectes.
- **Implementació**. En aquest capítol donarem a conèixer *com* s'ha construït l'aplicació, les classes i mètodes implementats que siguin més significatius per a la comprensió del funcionament del videojoc.
- **Resultats**. En aquest capítol mostrarem proves d'execució de l'aplicació, mostra imatges del videojoc, incloent interfícies, imatges de la partida, i tot allò que es pugui visualitzar del que s'ha implementat.
- **Conclusions**. Aquí exposarem les conclusions extretes un cop finalitzat el projecte, tant aquelles comunes com les personals que en traiem cada un de nosaltres.
- **Treball futur**. En aquesta secció pretendrem exposar allò que pensem que podria millorar el nostre videojoc, o bé ampliar-lo de manera interessant, i que per qüestions de planificació i temps no s'han tingut en compte.
- **Agraïments**. Aquí volem fer un petit homenatge a tota aquella gent que ens ha ajudat en un punt o altre del projecte, en major o menor mesura, donant les gràcies per l'ajuda desinteressada rebuda.
- **Annex**. Finalment, en aquest capítol hi posarem les referències bibliogràfiques que tinguem, així com esmentarem tutorials o manuals que ens han servit d'ajuda durant el transcurs del projecte.

1.5. Separació de tasques

En aquest apartat explicarem com s'han separat les tasques no comunes. Es dona per entès que totes les tasques que no estiguin aquí esmentades son comunes, i desenvolupades en equip i no individualment.

Nota important:

Inicialment es volien crear dues documentacions separades, una per a cada membre

del grup, on figuressin els objectius, tasques i desenvolupaments individuals. Durant el transcurs d'escriure la memòria s’ha vist que molts dels conceptes i de les funcionalitats dels dos membres del grup eren difícilment separables. És per aquest motiu, degut a aquesta estreta vinculació, que finalment s’ha pres la decisió de realitzar una única memòria conjunta.

El motiu principal que ens ha portat a unir les dues memòries ha estat facilitar la comprensió del treball fet als lectors d’aquesta memòria.

1.5.1. Pere Fonolleda

En Pere Fonolleda s'ha ocupat de les següents tasques:

1.5.1.1. Implementació de l'animació per óssos

Aquesta tasca inclou:

- L'edició amb Blender dels models utilitzats per tal de crear les animacions.
- La modificació de les classes de personatges per tal d'incloure-hi les animacions i les estructures d'óssos.
- La modificació de la classe del controlador d'entrada, així com totes les accions derivades d'aquest, per a que tingui en compte les animacions, les activi, desactivi, o faci les comprovacions pertinents.

1.5.1.2. Implementació de la interfície gràfica

Aquesta tasca inclou:

- Integrar les llibreries del *GBUI*, així com la creació de la classe **GbuiGameState** i la gestió del *GBUI* en tots els estats.
- Comporta la creació de totes les classes que hereten del **GbuiGameState**, amb excepció de la classe **Partida**, que tant sols ha estat modificada per a tenir en compte la GUI.
- L'edició d'estils i imatges que componen tots els menús.

1.5.1.3. Implementació del multijugador

- Això comporta el fet de tenir en compte no tant sols un controlador d'entrada, sinó varis, i que aquests es comportin correctament amb els personatges que controlen.
- Per tant, és una ampliació del controlador d'entrada construït conjuntament, adaptant-lo al multijugador.

1.5.2. Ferran Font

En Ferran Font s'ha ocupat de les següents tasques:

1.5.2.1. Implementació de la física

Aquesta tasca inclou:

- Integrar el motor físic *Physics*, i implementar-lo en els objectes que composen un escenari per tenir en compte les col·lisions.
- Això vol dir, que les classes **Personatge**, **PersonatgeNJ**, **Bustia**, **Escenari**, **FurgonetaDeCorreos** i **Paquet** tenen tota una part creada i desenvolupada només per a la interacció amb la física general.

1.5.2.2. Implementació de la música

Aquesta tasca inclou:

- La creació o obtenció d'arxius sonors i la seva edició amb el programa Audacity. A més, suposa gestionar la música a nivell ambient, tenint en compte que treballem amb música en 3D, i gestionar els efectes sonors que s'han d'activar o desactivar en certs esdeveniments de l'aplicació.

1.5.2.3. Implementació de la intel·ligència artificial

Aquesta tasca inclou:

- Que la classe **FurgonetaDeCorreos** s'autogestioni en quan a la càrrega i descàrrega de paquets es refereix, i els moviments d'entrada i sortida relacionats.

- També comporta el moviment per l'escenari dels elements de la classe **PersonatgeNJ**, la seva cerca de personatges jugadors i posterior persecució, o no persecució, segons criteris de distancia de visió o velocitat.

2. Estudi previ

En aquest apartat explicarem les tasques desenvolupades durant les primeres etapes de la realització d'aquest projecte. Hem hagut de fer una cerca per escollir aquell motor de jocs que més s'adaptés a les nostres necessitats, i a més hem hagut d'aprendre alguns conceptes teòrics de la programació d'aplicacions gràfiques en 3D.

2.1. Introducció als motors de videojocs

Al món del desenvolupament de videojocs i aplicacions complexes emprant gràfics és molt important no haver-se d'enfrontar directament amb una llibreria gràfica com pot ser OpenGL o DirectX. Tot i disposar de molta llibertat per a utilitzar-ne una, està clar que ens enfrontarem amb més problemes apart dels gràfics com poden ser un motor físic, importar models gràfics o la intel·ligència artificial. Així doncs, és lògic que es vulgui treballar amb un sistema a més alt nivell, i és aquí on entren en joc els motors de videojocs.

Per tant, un motor de videojocs (conegut en anglès com a *Game Engine)* fa referència a una aplicació de programari que permet el disseny, la creació i la representació d'un videojoc d'una manera més simple.

Es fa servir la paraula motor per una analogia amb els automòbils, ja que com el mecànic, aquest motor està sota el capó i no és visible però transporta el vehicle, és a dir, és el que fa la feina. Així doncs, els motors de videojocs, són els que, des d'un nivell més baix, s'ocupen de manejar allò que veiem, com les textures, els models, etc.

Podríem definir un motor de videojocs com a un recull d'eines especialment preparades per a confeccionar videojocs, que s'ocupen de les tasques a més baix nivell, per a que el programador que hi treballa pugui utilitzar unes funcions a un nivell més alt.

Hi ha vàries raons per les quals l'ús dels motors de videojocs és important. Algunes d'elles, son:

- Facilita el desenvolupament.
- Obre noves oportunitats de negoci, com és la venda de les llicències dels motors per a la realització d'altres jocs.
- Abstracció de la plataforma. Si un motor corre en vàries plataformes, el nostre joc també.
- Separació de motor i continguts. Això permet tenir varis grups de treball en paral·lel.
- Beneficis a tercers. Ja que els avanços en el motor poden beneficiar a varis jocs.
- Permet emfatitzar en la part artística, ja que al tenir més grau d'abstracció, es té menys càrrega de treball en programació.
- Major modularitat i reaprofitament.

Esquema típic d'un motor de videojocs:

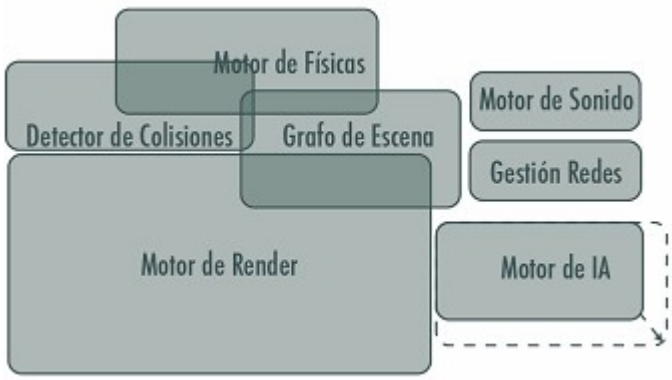


Figura 4: Esquema de les capes d'un motor de videojocs.

Com es pot veure en la Figura 4, la part de més pes en el nostre sistema és el motor de renderització, la part principal del motor gràfic. Això va lligat un a graf d'escena que és el que s'ocuparà de l'estructura de dades que afegim a l'escenari. Lleugerament unit a això, i al fet de que el nostre escenari ha de ser mòbil, ens trobem amb la detecció de col·lisions, que farà servir un motor de física per decidir com es mouran els nostres objectes.

Apart, disposarem de la possibilitat d'un motor de so, d'un motor de xarxa i d'un motor d'intel·ligència artificial, aquests independents dels gràfics, que ens acabaran de donar les funcionalitats que necessitem.

2.2. Diferències entre un motor gràfic i un motor de videojocs

Avui en dia és difícil parlar de la frontera que separa un motor gràfic d'un motor de videojocs. A nivell molt teòric, un motor gràfic es centra exclusivament en la representació de gràfics, mentre que el motor de videojocs implementarà altres mòduls habituals en les tasques que du a terme un videojoc.

A la pràctica ens trobem que aquesta barrera cada cop s'ha debilitat més, ja que la majoria de motors gràfics ja es consideren motors de videojocs al incorporar, encara que sovint siguin més senzilles, llibreries que s'ocupen de tasques com el so o la física. A més, si no les porten incorporades, és molt senzill adaptar noves llibreries en aquests motors gràfics.

De fet, els motors de videojocs acostumen a basar-se en un motor gràfic que després és ampliat. Afegeixen una sèrie de llibreries que estenen les funcionalitats dels motors base, i al instal·lar-los per a treballar ja es disposa d'aquestes eines completament integrades.

2.3. Cerca de motors de videojocs

A continuació mostrem un petit recull de les opcions que es van estudiar abans de l'elecció del motor escollit.

Al principi, vam estudiar la possibilitat de treballar sobre un motor gràfic i connectar-hi els mòduls addicionals que creguérem necessaris.

Tots els motors llistats a continuació son aquells que vam considerar possibles per a la implementació del nostre videojoc. Apart, durant la nostre cerca vam trobar una sèrie de motors o frameworks interessants que vam descartar directament pel fet de no tenir una llicència lliure, i per tant no estan llistats a continuació.

2.3.1. Ogre3D

OGRE (Object Oriented Graphics Engine) és un motor de gràfics en tres dimensions multiplataforma. Això és possible gràcies a que està construït de manera que no es compromet amb una API en particular, ja que el motor suporta tan l'ús del DirectX com d'OpenGL. A més, la principal avantatge de l'Ogre sobre altres engines 3D és que es un projecte open-source sota llicència LGPL.

Ogre és el motor de codi lliure més utilitzat, ja que és un projecte molt consolidat, potent, i adaptat a totes les plataformes.

Les seves característiques generals son:

- Disseny orientat a objectes. Interfície simple i fàcil d'utilitzar dissenyada per a que requereixi poc esforç el renderitzat d'escenes en tres dimensions.
- Arquitectura basada en plugins molt flexibles que permeten ampliar les funcionalitats del motor.
- Disseny net i ben documentat de les classes del motor.
- Independentment de l'API gràfica es pot utilitzar OpenGL o DirectX.
- Té una comunitat d'usuaris molt gran i molt compromesa.

Figura 5: Imatge del joc comercial Ankh

El llenguatge de programació sobre el que està escrit aquest motor és C++.

Un projecte comercial del motor Ogre3D, és el videojoc *Ankh* (observar Figura 5).

Pàgina web: http://www.ogre3d.org

2.3.2. Crystal Space

Crystal Space és un framework per al desenvolupament d'aplicacions 3D escrites en C++. Va ser publicat l'any 1997 sota la llicència LGPL.

Crystal Space s'utilitza típicament com un motor de videojocs però el framework es més general i pot ser utilitzat per a qualsevol tipus de visualització en 3D.

És molt portable i s'executa en Windows, GNU/Linux, UNIX i Mac OS X. Pot utilitzar opcionalment OpenGL en totes les plataformes, SDL també en totes les plataformes, X11 i SVGALib en UNIX o GNU/Linux i també pot utilitzar rutines d'assemblador mitjançant el NASM i l'MMX.

Tot i ser un motor molt competent, té una comunitat d'usuaris significativament menor que la d'Ogre, a més, té una corba d'aprenentatge molt alta.

Figura 6: Imatge del MMORPG PlaneShift creat amb CrystalSpace

Un dels projectes més importants d'aquest motor, és el desenvolupament del videojoc *PlaneShift* (Figura 6).

Pàgina web: http://www.crystalspace3d.org/

2.3.3. Irrlicht

Irrlicht és un motor de videojocs lliure i gratuit, publicat sota una llicència open-source basada en la llicència zlib/libpng, el 2006.

És multiplataforma i funciona sobre Windows, Mac OS, GNU/Linux i Windows CE, i degut a la comunitat que té s'han creat versions per a Xbox, PSP, Symbian OS i iPhone.

Està originalment creat per a funcionar sota C++, però també és compatible amb altres llenguatges com el .NET, Java, Perl, Ruby o Phyton.

Té una sèrie d'extensions que permeten ampliar les seves funcionalitats però que no venen per defecte en el motor. L'avantatge d'aquestes extensions és que en la seva majoria son creades per desenvolupadors de l'Irrlicht, i és molt senzill afegir-les.

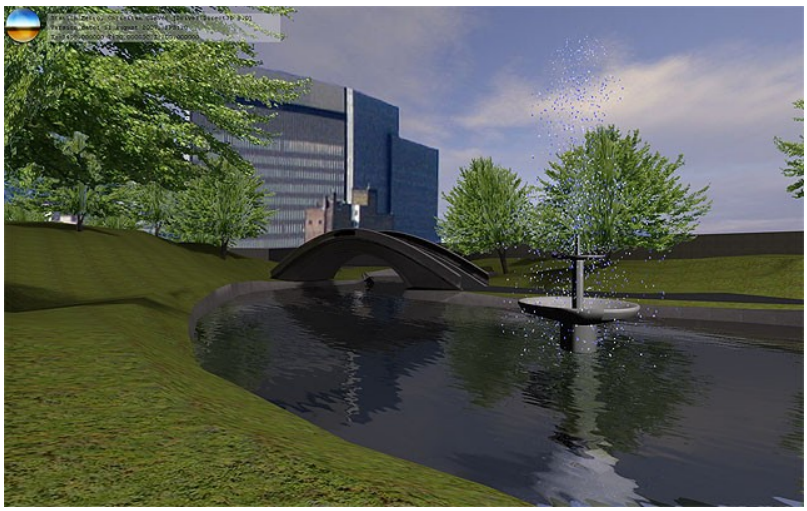


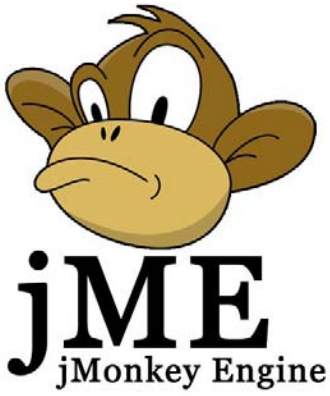
Figura 7: Render de demostració d'efectes de reflex i partícules del motor

Com es veu a la Figura 7, el motor de renderitzat del videojoc és molt complet.

Té una comunitat bastant àmplia i molt activa.

Pàgina web: http://irrlicht.sourceforge.net/

2.3.4. jMonkey Engine



Serious Monkey, Serious Engine.

El jMonkey Engine és un motor de videojocs 3D d'alt nivell escrit completament en Java. Aquest engine treballa sobre OpenGL a través de la coneguda llibreria LWJGL (Lightweight Java Game Library), però millora àmpliament les possibilitats d'aquesta. A més, està en desenvolupament el suport directe via JOGL (en anglés, JAVA OPENGL. Llibreria que permet accedir a OpenGL mitjançant programació en Java).

jME va ser publicat per primera vegada 2003 sota llicència BSD. Aquest motor està sent utilitzat per estudis de desenvolupament de jocs comercials, així com en algunes universitats s'utilitza en classes relacionades amb la programació de videojocs.

Hi ha bastants jocs desenvolupats o bé en desenvolupament sobre aquest motor, anant aquests des de bàsics puzzles a jocs més complexes com per exemple alguns MMORPG[1] de qualitat gràfica molt interessant, com l'Spirits, que es pot veure a la Figura 8.

Actualment es troba en la versió 2.0 de manera estable, i ja s'està treballant en una versió 3.0 degut als alts estàndards de la nova generació de hardware.

1 De les sigles en anglès de Massively Multiplayer Online Role-Playing Game, que vol dir Joc de Rol Multijugador Massiu en Línia.

Figura 8: Imatge del joc Spirits

Una de les raons per les que aquest motor és interessant, es perquè té una sèrie de llibreries i motors addicionals que son molt fàcilment enllaçables. A més, entre aquests motors es troben les adaptacions a Java de coneguts sistemes com el Physics o l'OpenAL.

Un altre aspecte interessant es que al ser natiu de Java, permet la creació d'aplicacions multiplataforma que, a més amb els navegadors actuals, fàcilment es poden adaptar dintre d'un applet d'aquests, i també es pot utilitzar mitjançant el protocol de llançament d'aplicacions de Java mitjançant Internet (JNLP).

Pàgina web: http://www.jmonkeyengine.com/

2.3.5. Yake

Yake és un petit, flexible i multiplataforma game engine. Llicenciat sota LGPL i llicències pròpies.

Aquest framework per a videojocs està basat en el motor gràfic Ogre, i a més ens proveeix de varis nuclis funcionals i una capa d'abstracció de l'API per als usos més comuns, així com àudio, físiques, físiques per a la utilització en xarxes,

pathfinding (algoritmes d'intel·ligència artificial calculadors de ruta), etc.).

Té una comunitat d'usuaris més aviat petita, les seves actualitzacions no son gaire constants, i a més, ens vàrem trobar en que la gran majoria d'usuaris d'aquest engine treballaven sobre la plataforma Windows, i no ens podien ajudar massa en l'utilització sobre GNU/Linux.

Pàgina web: http://www.yake.org

2.3.6. OGE

Open Game Engine és un projecte per a crear un complet motor de videojocs que inclogui les gràfiques, les físiques, els controladors d'entrada (tant sigui de teclat, com de ratolí o GamePad), controladors de xarxa i scripting entre d'altres.

Com en el cas del motor anterior, aquest framework està basat en el motor Ogre, però expandeix les seves possibilitats.

La part més interessant d'aquest motor és que se l'hi ha desenvolupat un component, anomenat OGEd (Open Game Engine Editor, es pot veure a la Figura 9), que permet la visualització d'escenes en una finestra apart per a poder visualitzar allò que fem mentre treballem.

Aquest projecte és relativament nou, va començar el gener del 2006 i va ser publicat amb llicència LGPL.

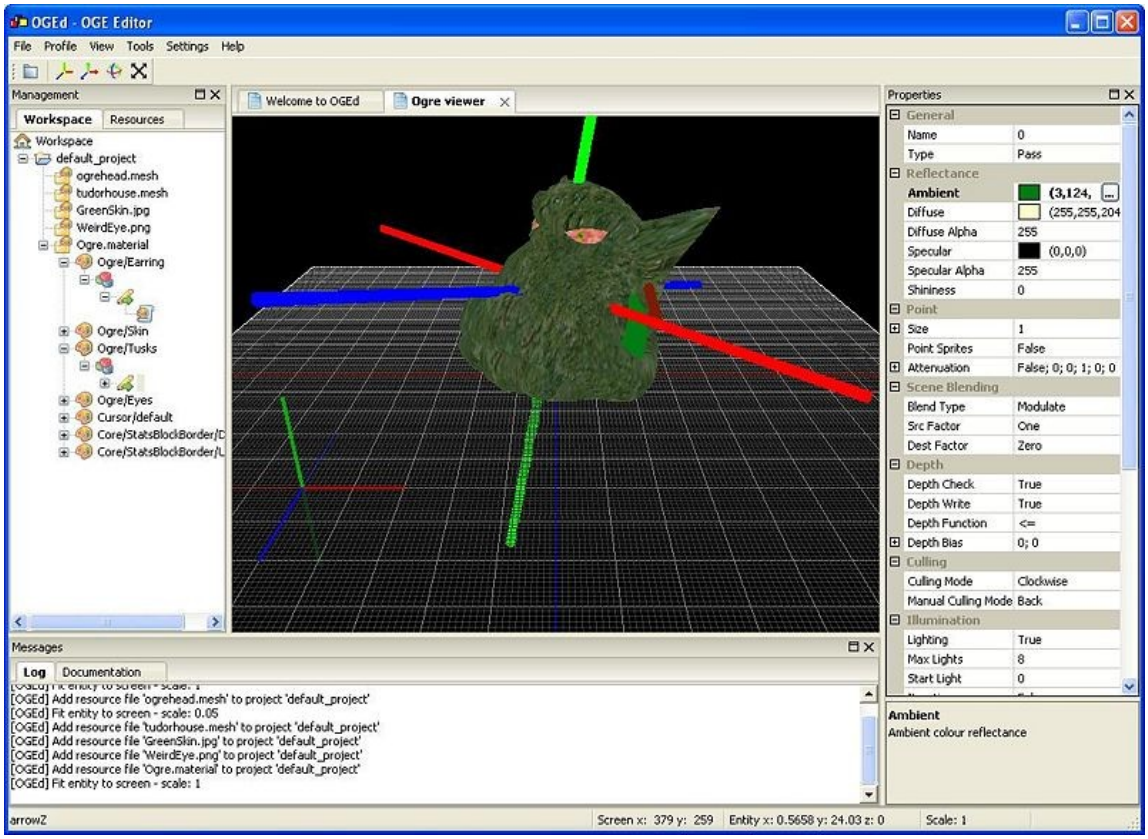


Figura 9: Captura de pantalla del plugin OGEd

Pàgina web: http://www.opengameengine.org

2.3.7. Raydium

Raydium és un motor de videojocs senzill que apunta, al contrari d'altres motors com són Ogre o Crystal Space, a un disseny ràpid i senzill, però a la vegada competent. Al principi estava planejat com a una petita llibreria 3D per a practicar amb OpenGL, però va començar a gestionar més i més coses fins arribar a ser un complet motor de videojocs.

Com a exemple, la gent de Raydium ens ensenyen el joc New Sky Driver, un simple joc creat amb menys de 750 línies de codi (Figura 10).

Tot i ser senzill, Raydium incorpora funcionalitats com:

- Portabilitat entre sistemes operatius gràcies al ANSI C.
- Renderitzat en OpenGL: suport per boira, il·luminació dinàmica, transparencies, SkyBoxes, multitextures, etc.

- Una API molt simplificada amb una corba d'aprenentatge molt simple.
- Integració amb el motor de físiques ODE (Open Dinamic Engine).
- Consola en el joc, permetent accedir a Raydium o a una aplicació en temps real.
- Llenguatge d'scripting en PHP.
- API de xarxa integrada per a jocs multijugador.
- Suport OpenAL amb posicionament de fonts 3D.
- Importació i exportació de models en formats 3D.
- Suport per a joysticks i altres complements de joc. Actualment només en Linux.

Figura 10: Imatge del joc NewSkyDriver creat amb Raydium

Raydium és software lliure sota llicència GPL.

Pàgina web: http://raydium.org

2.3.8. OpenSceneGraph

Motor lliure i gratuït publicat sota llicència LGPL. Està escrit en C++ i treballa sobre OpenGL, cosa que el fa compatible amb Windows, Linux, Mac OS X, IRIX, Solaris i FreeBSD. S’ha utilitzat en diferents camps: videojocs, modelatge, realitat

virtual, visualització científica, etc. Tot i això no disposa d’una gran comunitat.

Pàgina web: http://www.openscenegraph.org

2.3.9. Delta3D

És un motor de jocs lliure sota llicència LGPL. La seva característica principal és que, gràcies al seu disseny modular, pretén integrar altres projeces de codi lliure com OpenSceneGraph, Cal3D o OpenAL sota una API fàcil d’utilitzar.

Pàgina web: http://www.delta3d.org

2.4. Elecció del jMonkey Engine

En aquest apartat exposarem les raons per les quals vam decidir que el jMonkey Engine seria l'eina que finalment utilitzaríem en el nostre projecte.

Primer de tot, aquesta eina està llicenciada sota la llicencia BSD, la qual és software lliure. Això, més que una raó per a decidir-nos, era un requisit indispensable.

Un altre motiu per a triar aquest engine va ser que aquest funciones sobre Java, i fos fàcilment executable des de qualsevol plataforma que suporti Java mitjançant un navegador web. Això és important perquè, tot i que nosaltres només treballem amb software lliure, volíem poder fer arribar el joc a qualsevol plataforma de codi privatiu, com Mac OS o Windows. Si bé es cert que altres motors tenen la possibilitat de ser compilats en qualsevol d'aquestes plataformes, això obliga a tenir la plataforma en qüestió sobre la que vols compilar, i no és compatible amb la nostre idea inicial del projecte de treballar només sobre programari lliure.

A més, com que hem estat treballant el Java durant la carrera, el temps per a dominar el llenguatge es podria considerar nul. Per tant, només ens havíem de preocupar de la corba d'aprenentatge del framework; que veient els testos d'exemple del motor, podem veure que és senzilla, ja que inclou classes bàsiques (sota el prefix Simple) que fan la majoria de feina bàsica per nosaltres, i a més, quan ja es coneix el funcionament millor, es pot passar a classes més complexes que donen més control sobre l'aplicació.

També era important una bona comunitat, tant d'activitat com de documentació; que en els dos casos es pot considerar acceptable. Si bé és cert que no tenia una comunitat tan gran ni tanta documentació com el projecte Ogre3D, en tenia la suficient com per a tenir la certesa que en la major part dels casos, s'obté una resposta relativament ràpida a una pregunta.

Finalment, un últim punt que va ajudar a la decisió, va ser el fet de que integrava tots els mòduls/motors auxiliars que necessitàvem. En aquells casos que no els integrés, tenia una fàcil integració correctament documentada a la seva wiki.

2.5. Descripció d'alguns conceptes teòrics d'informàtica gràfica relacionada amb els videojocs

En aquest apartat ens disposem a explicar molt breument alguns conceptes bàsics que cal conèixer de la informàtica gràfica per a poder utilitzar un motor de gràfics, inclòs en el nostre motor de videojocs.

Com que la gestió d'aquests elements a nivell de programació canvia en cada motor, explicarem els conceptes a nivell teòric descriptiu per a saber de que es tracten els objectes que en el futur farem servir.

2.5.1. Vector

En la física un vector (veure Figura 11) és un concepte matemàtic que s'utilitza per descriure magnituds tals com velocitats, acceleracions o forces, en les quals és important considerar no només el valor sinó també la direcció i el sentit. Es representa per un segment orientat per denotar el seu sentit, la seva magnitud (la longitud de la fletxa) i el punt d'aplicació.

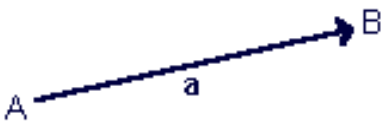


Figura 11: Representació d'un vector en el pla

En el cas de la informàtica gràfica els vectors estan situats en un espai vectorial de

tres dimensions, i per tant es representen amb tres coordenades.

2.5.2. Bone (Ós)

En la biologia, L'os és un teixit conjuntiu, de notable elasticitat, lleuger i de gran duresa. Compost per cèl·lules especialitzades i fibres que formen una matriu. En els vertebrats efectua una triple funció: la de sosteniment del cos, la de protecció d'alguns òrgans (cervell, cor, pulmons) i la de possibilitar el moviment (a tall de palanques mogudes pels músculs).

En la informàtica gràfica, utilitzem el concepte d'ós per a animar els personatges.

En l'animació de personatges, un ós no es res més que una estructura que té una influencia sobre la malla (que seria la pell) d'un model simulant el comportament real, podent fer que les malles es deformin per allà on els óssos conflueixen i poden exercir més o menys index de deformació en aquests punts. Anomenarem esquelet (o Skeleton en anglès) al conjunt d'óssos que animen per complet a un personatge.

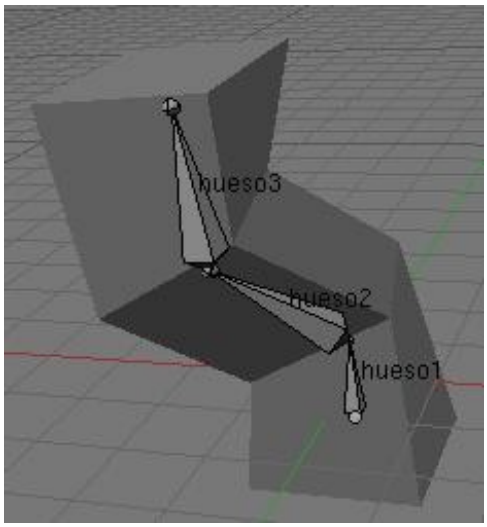


Figura 12: Exemple d'una estructura d'óssos amb deformació de malla

2.5.3. Caixa envolupant (BoundingBox)

Es coneix com a caixa envolupant (BoundingBox en anglès) aquell requadre que tanca amb el mínim espai possible la totalitat d'un objecte. D'aquesta manera convertim una figura complexa amb un cub, i així simplifiquem molt la tasca de saber quan aquesta figura entra en contacte amb altres elements de l'escena.

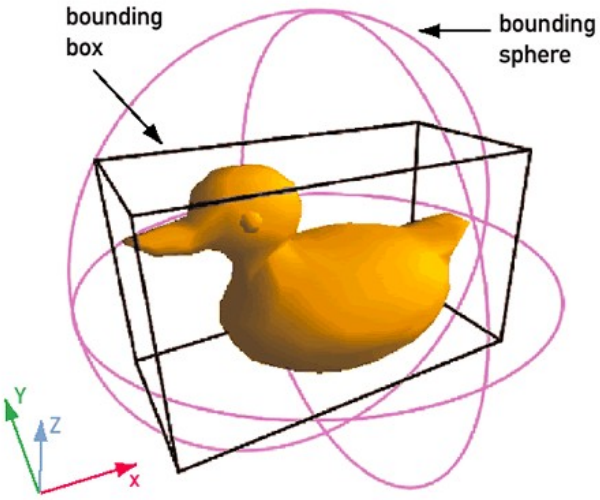


Figura 13: Imatge representativa de BoundigBox en cub i esfera

Apart de les BoundigBox, també hi ha elements geomètrics envolupants com ara l'esfera o el cilindre com es pot veure a la Figura 13.

2.5.4. Skybox

Un Skybox (literalment caixa de cel) és un métode per a crear de manera senzilla una imatge de fons per fer que l'escena del videojoc tingui una aparença més gran del que realment és.

Aquesta escena està tancada en un cub en el qual cel, muntanyes, edificis llunyans i altres objectes son mostrats en cada cara del cub, creant una il·lusió en tres dimensions.

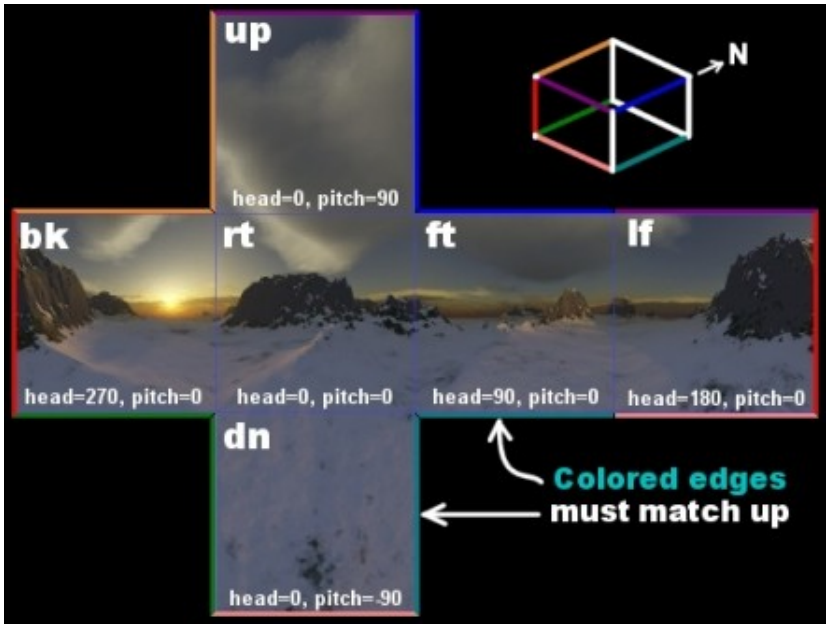


Figura 14: Imatge que ens mostra com funcionen els Skyboxes

Es fa servir perquè processar el 3D d'un videojoc és computacionalment car, específicament en els jocs en temps real, per tant no podem estar renderitzant escenaris que estan molt lluny de la càmera. D'aquesta manera, amb un simple cub amb sis textures col·locades a cada cara donen la il·lusió òptica d'un mon 3D en temps real com es pot veure a la Figura 14.

3. Eines utilitzades

En aquest capítol s’esmenten les llibreries i programes utilitats per tal de dur a terme aquest projecte. Es mostraran les eines que han permès la implementació de l’aplicació, tant les més bàsiques com aquelles que ens hagin servit de suport, així com els programes necessaris per construir la documentació corresponent.

Cal esmentar que totes les eines utilitzades, així com llibreries i demés programari, són software lliure, sota diferents llicències però que garanteixen que tots tinguem dret a accedir i treballar amb aquest software.

3.1. Sistema operatiu

El sistema operatiu triat ha sigut la distribució de GNU/Linux Ubuntu. Començant en la versió 8.04 al principi del projecte, fins a la versió 9.04 en el tram final. Aquest sistema operatiu és una derivació de la distribució GNU/Linux Debian.

Hem triat aquest sistema operatiu perquè, a més de ser el que diàriament utilitzem nosaltres, i per tant ens és molt còmode per treballar, ofereix compatibilitat amb tots els programes que necessitem per el desenvolupament, estabilitat i seguretat com a sistema operatiu i un bon rendiment de la màquina, sense carregar-la de processos innecessaris.

3.2. Llibreries utilitzades

A continuació explicarem les llibreries que utilitzarem per desenvolupar el nostre videojoc.

3.2.1. jMonkey Engine

3.2.1.1. Introducció al jMonkey Engine

Comentats anteriorment els punts generals d'aquest motor, passarem a explicar la seva arquitectura interna i funcionament a nivell més baix.

jMonkey Engine va ser construït per a omplir la falta de motors gràfics complerts escrits en Java. A grans trets, podem dir que aquest motor treballa sobre un graf

d'escena d'alt rendiment basat en un API de gràfics. La seva estructura està basada en el llibre 3D Game Engine Design de David Eberly.

En quant a la renderització respecta, fent servir una capa d'abstracció, aquest motor permet la connexió amb qualsevol d'aquests sistemes.

3.2.1.2. Funcionament del jMonkey Engine

La principal classe d'aquest motor serà una derivació de la classe pare AbstractGame. Com el seu nom indica, aquesta és una classe pare abstracta que serà derivada en classes fills (fins a nombrosos nivells d'extensió). Tota aplicació de jME haurà d'instanciar almenys una d'aquestes classes que serà la que s'ocuparà de totes les funcions bàsiques del motor de videojocs, com ara l'actualització de dades, renderització de l'escena, inicialització de variables, etc.

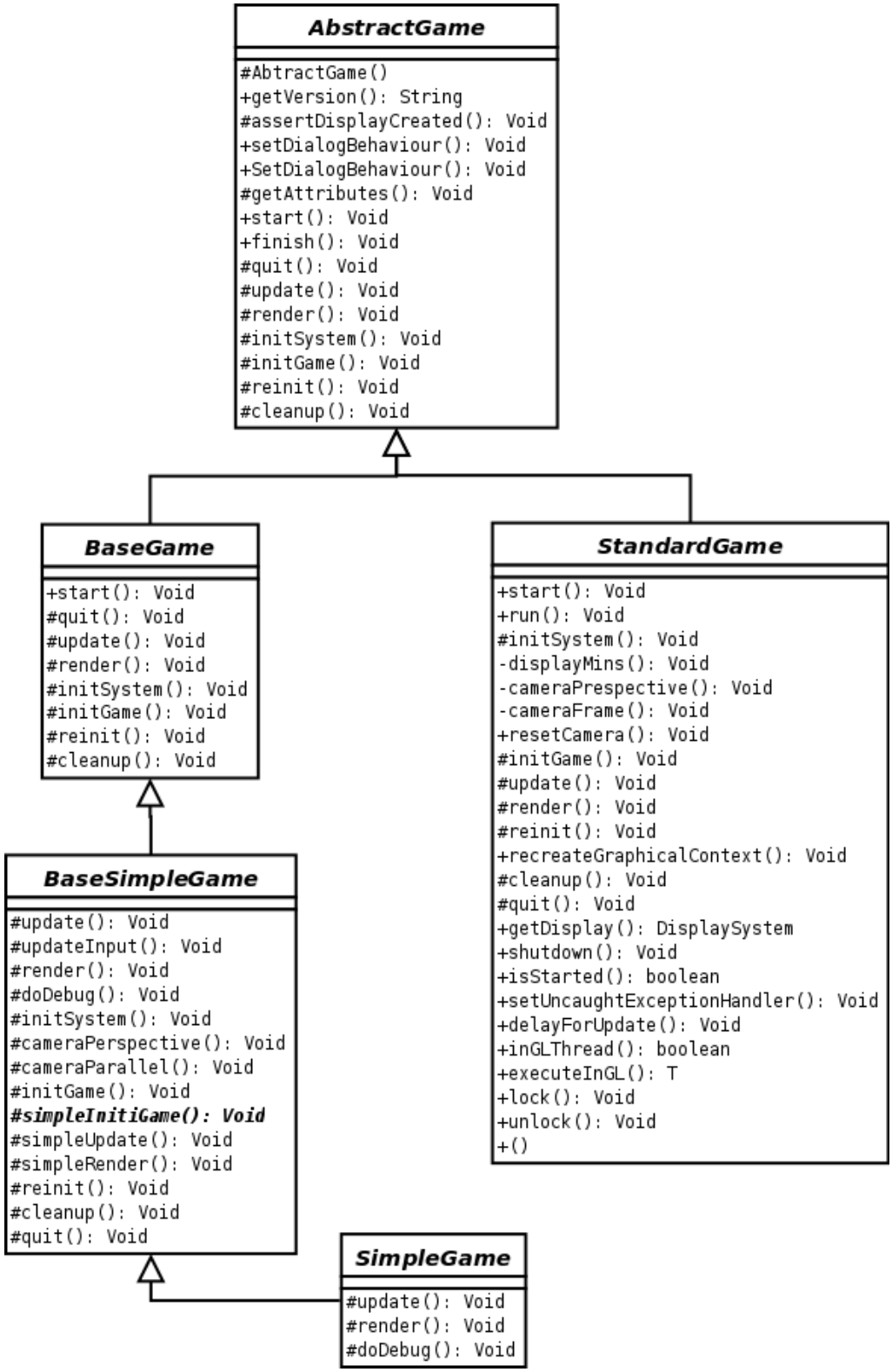


Figura 15: Diagrama de les classes bàsiques que hereten de AbstractGame

Inicialment, tal i com veiem a la Figura 15 estendrem de SimpleGame per tal de que aquesta classe sigui l'encarregada de fer la majoria de funcions comunes sense que

nosaltres ens haguem de preocupar massa del seu funcionament, sinò que donem tot el poder a la classe per a gestionar-ho. D'aquesta manera, comencem a entrar en contacte amb el motor de videojocs i la seva estructura de nodes, malles, llums, etc. Més endavant ja passarem a gestionar personalment altres facetes com son l'entrada de teclat, ratolí o controlador de jocs (Input), o l'actualització d'algunes dades amb classes de més baix nivell com BaseGame o StandardGame.

Dintre d'aquestes dos classes de més baix nivell que hem comentat, la principal diferencia es que en tots dos casos tenim un major control del motor, però en l'StandardGame, a més, s'implementa la interfície Runnable, que permet treballar amb fils d'execució, i una sèrie de classes del motor anomenades States que explicarem a continuació.

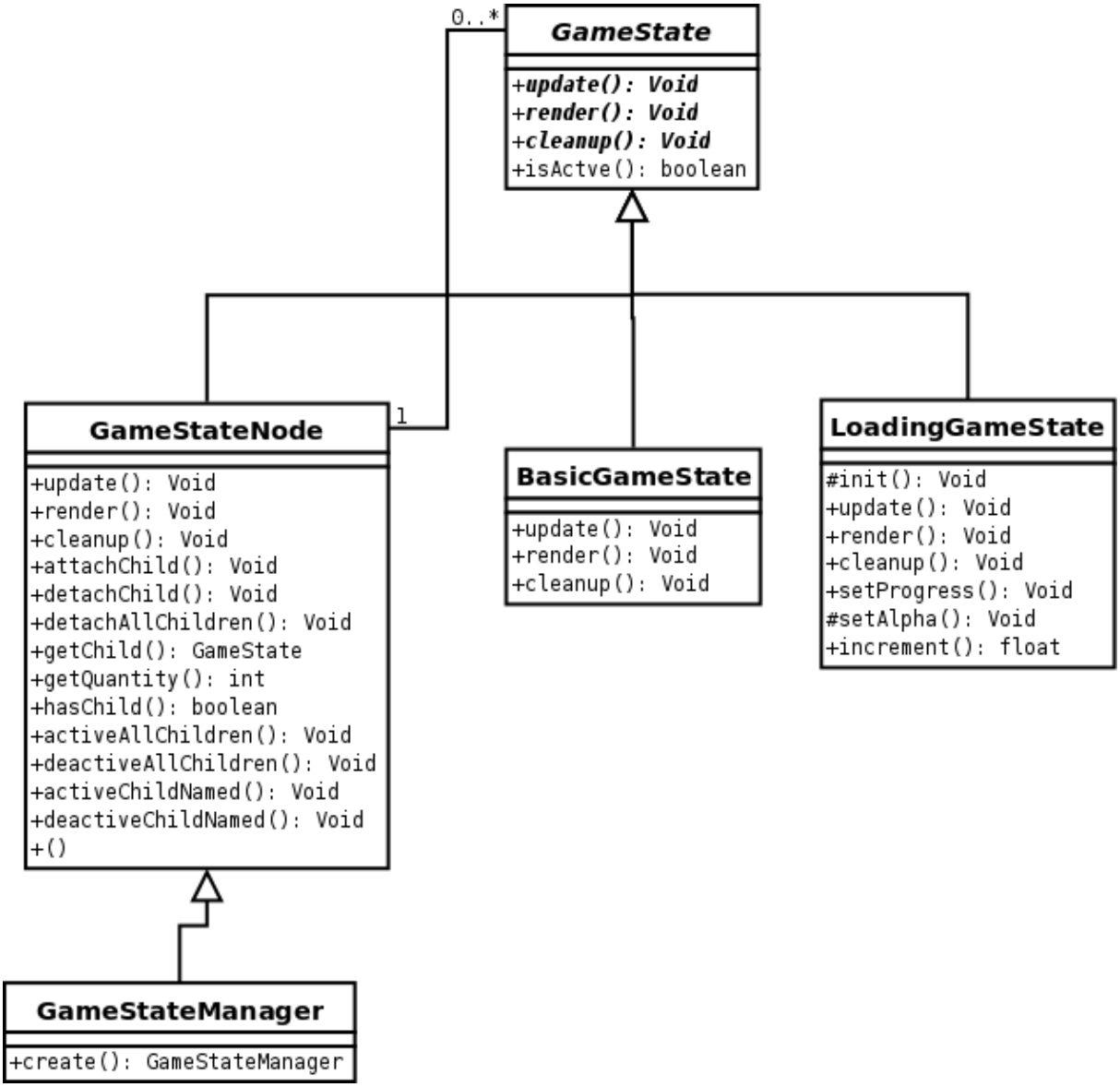


Figura 16: Diagrama de classes bàsic que hereta de GameState

En la Figura 16 podem veure el diagrama de les classes bàsiques que hereten de GameState que són més comunes en les aplicacions creades amb el motor jME. Nosaltres volem escollir aquesta manera de treballar ja que així gestionem la nostra aplicació per estats, a més que al fer servir fils d'execució, alliberem de càrrega computacional la màquina i hauríem de tenir una execució més fluida.

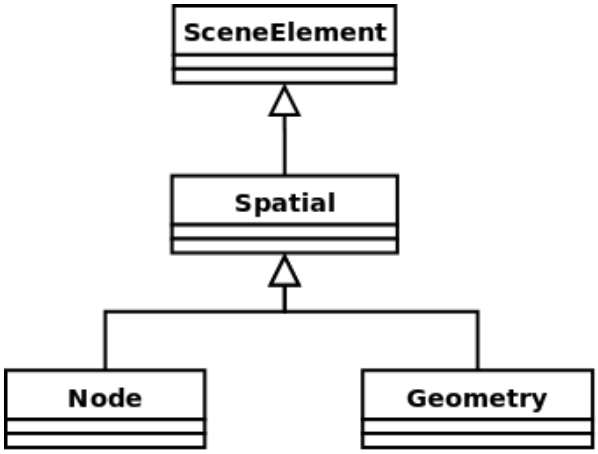


Figura 17: Diagrama bàsic de les classes principals dels elements de l'escena

Un cop tenim clar com començarà el nostre joc, és hora de saber com col·locar els elements.

Qualsevol element que nosaltres gestionarem serà fill d'un node o de la geometria. Com podem veure a la Figura 17, els dos són descendents d'Spatial, que, a la seva vegada, hereta d'SceneElement. Per tant, tot i que pel nostre nivell de concepció actualment separem nodes de geometria, en realitat tot son elements d'escena.

Entre aquestes dos classes, cal tenir clar que tenen un funcionament molt diferent. La classe Geometry té un funcionament de representació, mentre que la classe Node té un funcionament d'estructuració. Per tant, tots aquells elements que afegim a la nostra escena, dependran d'algun node pare, així que seguint l'arbre de nodes, cada escena tindrà un node arrel, al que anomenarem rootNode, del qual hi penjarà tot.

Per a que ens fem una idea, una estructura de nodes possible pel nostre joc podria ser (extreta de la mateixa documentació del jME) la mostrada a la Figura 18.

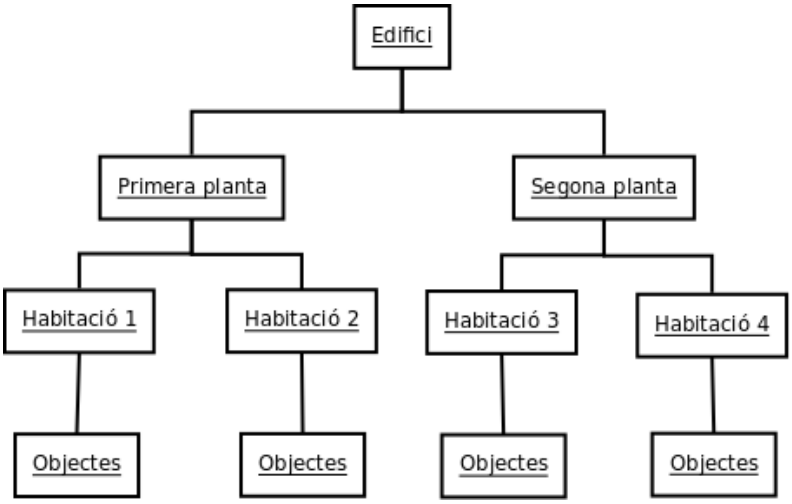


Figura 18: Exemple d'un possible arbre de nodes

En la Figura 18, veiem un arbre de nodes representat com si fossin elements d'un edifici. El nostre rootNode seria l'edifici d'on pengen la resta de nodes, els primers fills serien les plantes, després les habitacions, i després un node per a cada objecte que es trobés dintre. Remarquem que per a facilitar la comprensió, hem obviat els elements de geometria.

Si ens trobéssim, per exemple, a l'habitació 2, al fer el recorregut de l'arbre de nodes podríem veure que a l'hora de renderitzar l'escena tot el que no pengi de la Primera planta pot ser descartat, pel que tota la Segona planta serà descartada. A més, l'Habitació 1 també pot ser descartada segons l'arbre, i per tant tots els objectes també. Això es fa per a un millor rendiment del motor a l'hora de pintar les escenes.

3.2.2. Monkey World 3D

Monkey World 3D és un editor del graf de l'escena per al jMonkeyEngine. Permet editar els elements de l'escena, el terreny, les físiques i les animacions.

L'objectiu és poder crear un editor WYSIWYP (What you see is what you play), editor que crea els elements a la vegada que es va veient com queda a l'escena tal i com es pot veure a la Figura 19.

El Monkey World 3D es presenta com una sèrie de plugins per l'**Eclipse,** així que es pot escollir entre instal·lar-lo o descarregar directament un Eclipse amb els plugins

instal·lats.

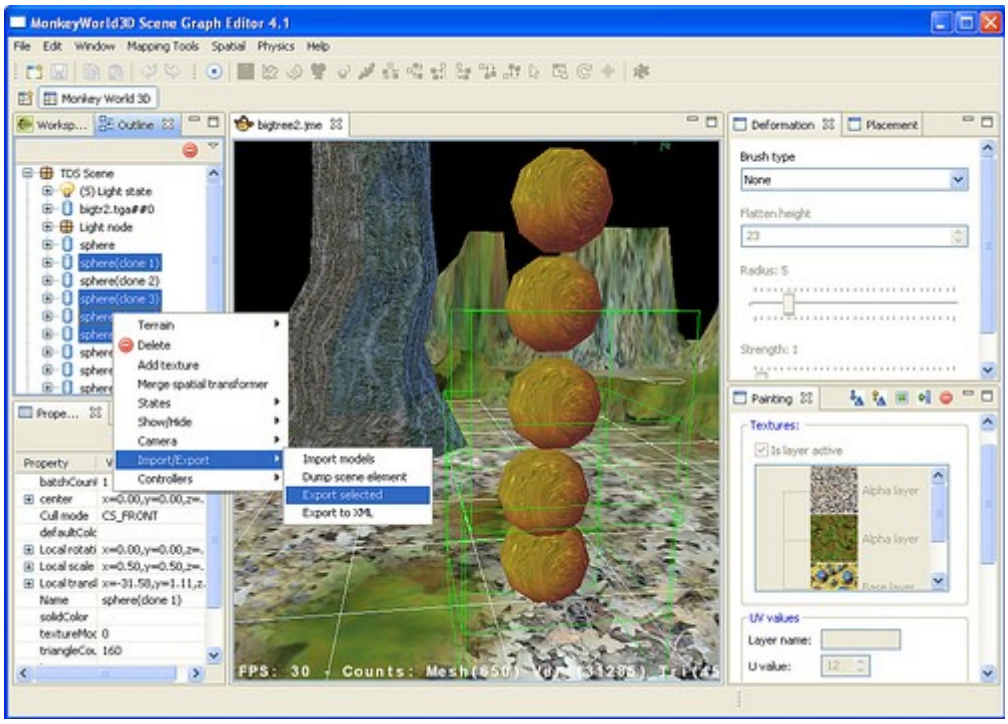


Figura 19: Captura de pantalla del Monkey World 3D

A l'hora d'exportar els escenaris al jMonkeyEngine ens crea un fitxer *.jme que podem obrir sense problemes.

Hem escollit aquestes llibreries per poder fer el disseny dels nivells, així com la col·locació d'obstacles, col·locació de textures, els SkyBoxes, entre moltes d'altres avantatges.

La llicència del Monkey World 3D és BSD.

Pàgina web: http://www.mw3d.org

3.2.3.jME 2 Physics 2

El JME 2 Physics 2 ens proveeix d'una interfície entre el JMonkey Engine i l'ODE (Open Dynamics Engine). Està basat en una lleugera modificació de l'OdeJava i ens proporciona una manera senzilla per crear un

món amb físiques i poder-hi afegir objectes. Un exemple senzill seria una caixa caiguen al terra, aplicació que s'escriuria amb molt poques línies.

Una de les característiques importants del JME Physics 2 és la vista de Debug de les Físiques. Col·lisions, articulacions, forces i velocitats son mostrades per pantalla com es pot veure a la Figura 20. Això és realment important per tal de tenir una visió del que realment està passant en la simulació, i així poder veure perquè les físiques tenen o no tenen el comportament desitjat.

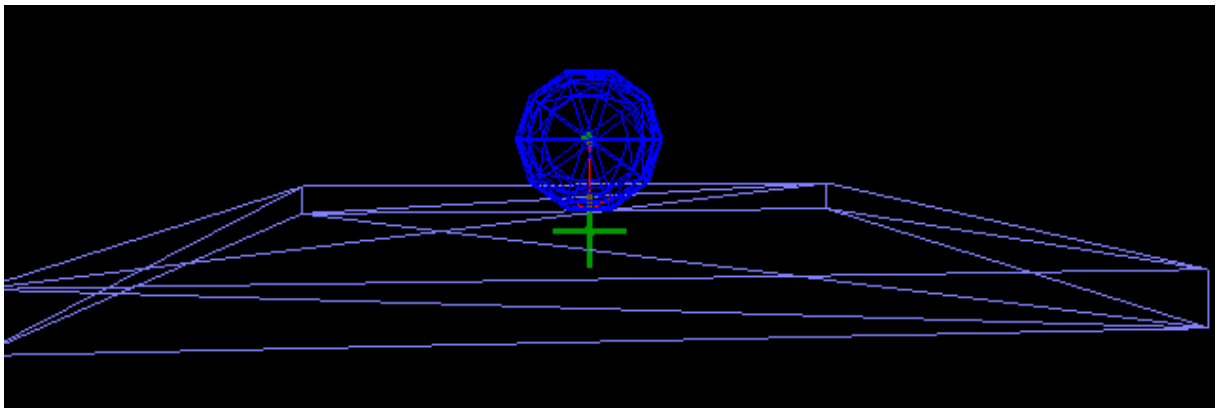


Figura 20: Vista del debug mode del motor de física Physics

Altres característiques:

- Podem reproduïr les físiques sense carregar la malla dels objectes.
- Carregar/Guardar representacions físiques.
- La vista de Debug mostra la representació de les físiques, fins i tot com actuen amb límits.
- La correcta manipulació dels nodes incrustats.
- Col·locació del centre de masses.
- Afegir un vector indicant cap a on actua la gravetat.
- Una bona integració amb el graf d'escena.

3.2.3.1. Funcionament

Un clar exemple del funcionament del JME 2 Physics 2 és tenir una habitació a on hi

hauria una taula i una pilota. Aquesta pilota podria moure's per l'habitació poden col·lisionar amb la taula. L'esquema es pot veure a la Figura 21.

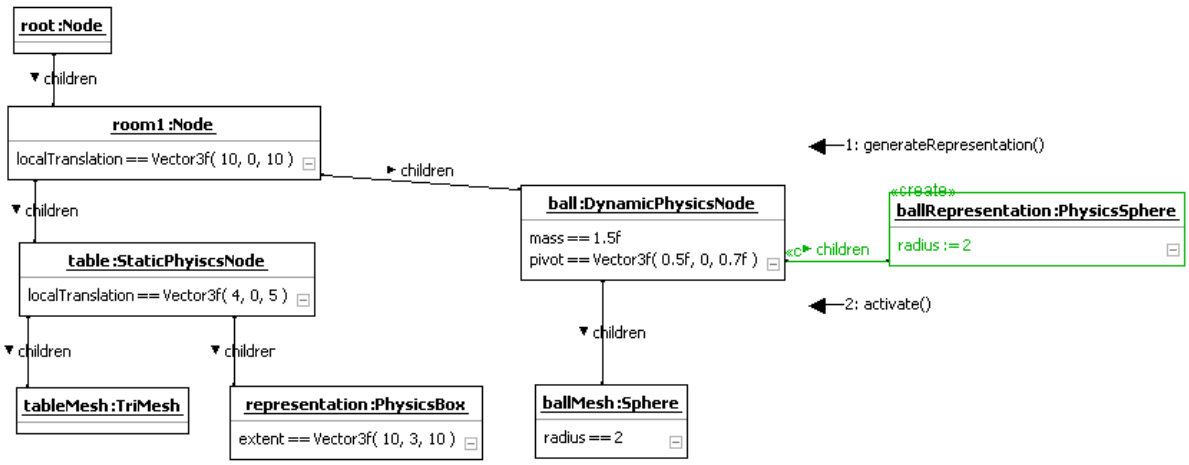


Figura 21: Esquema d'un graf d'escena amb nodes físics

En aquest exemple (Figura 21) es veu clarament la diferència entre els nodes dinàmics i els nodes estàtics. A més, també es veu que tot penja del *rootNode. D*'ell penja el node *habitació* i aquest últim és pare de la *taula* i de la *pilota.* Un node estàtic és aquell al que no se li aplicarà cap força, simplement interactuarà gràcies al contacte amb nodes dinàmics. Els nodes dinàmics són aquells als que aplicarem les forces. La taula és un clar exemple de node estàtic i la pilota de node dinàmic, ja que aplicarem la força a la pilota que rebotarà o no a la taula.

A part cada node físic, ja sigui estàtic o dinàmic, tindrà dos fills més: un que serà la seva representació gràfica i l'altre que serà la física. Aquests nodes són els que controlarà el motor Physics.

3.2.3.2. Classes principals

A continuació es mostrarà un diagrama de les classes necessàries per a introduir-se en el JME 2 Physics 2. A la Figura 22 és poden veure les classes principals i com estan relacionades entre elles. Només apareixen les classes més bàsiques i importants, que seran les classes que nosaltres treballarem.

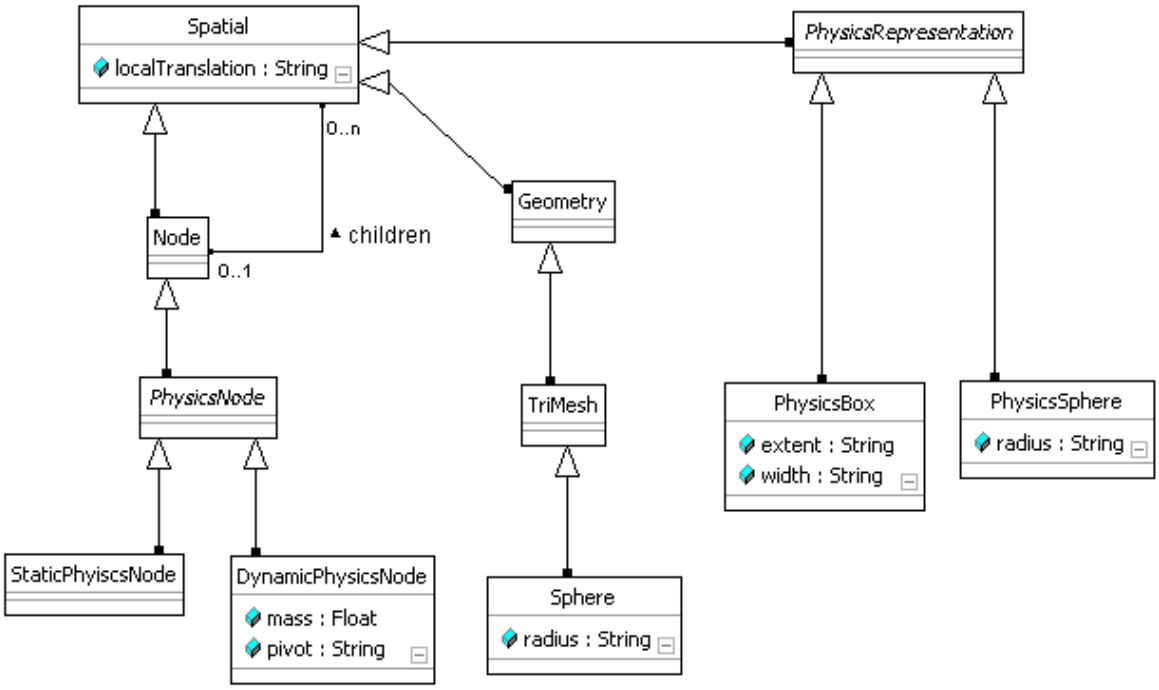


Figura 22: Diagrama de les classes més importants del Physics

En el diagrama de classes mostrat a la Figura 22, podem veure que tot hereta de la classe **Spatial**. Aquesta, explicada anteriorment, té dos classes que hereten d'ella, **Node** i **Geometry.** Totes aquestes classes formen part del motor jME. A partir d'aquí trobem la nova classe *PhysicsRepresentation* que també hereta de **Spatial.** D'aquesta classe hereteran les representacions físiques de les formes geometriques bàsiques, així com el cub, l'esfera, la càpsula, etc.

Per altra banda podem veure com de la classe **Node** hereta la classe **PhysicsNode**. D'aquesta classe hereteran dos classes més, els nodes físics estàtics (**StaticPhysicsNode**) i els nodes físics dinàmics (**DynamicPhysicsNode**). Aquests dos tipus de nodes seran els que el motor controlarà.

Els nodes estàtics com el seu nom indica, seran nodes als quals no podrem aplicar-hi forces, ni velocitats, etc. En canvi, als nodes dinàmics, nosaltres podrem aplicar-li forces segons un vector que contindrà la direcció i el mòdul.

Gràcies a aquestes llibreries incorporades a l'Eclipse podem treballar amb un motor de física.

Pàgina Web: https://jmephysics.dev.java.net/

3.2.4. GBUI

Les llibreries GBUI són unes llibreries utilitzades per la creació de la interfície gràfica d'usuari (GUI, de l'anglès Graphic User Interface).

Aquestes llibreries són una ampliació de les llibreries BUI (Banana User Interface), creades per al joc Bang! Howdy, que es pot veure a la Figura 23.

Figura 23: Imatge d'una captura de pantalla de joc Bang! Howdy

3.2.4.1. Funcionament

En termes generals, són una extensió de les classes **Swing** de Java estàndards, adaptades per a que treballin amb BSS, un format de fitxer propi semblant als fulls d'estil en cascada (CSS) creats per a poder donar estil als botons i demés elements de la interfície de manera fàcil i en bloc.

El que fem amb el GBUI és crear un objecte finestra de tipus **BWindow** que contindrà tots els nostres elements, podent ser aquests botons, texts, llistes, etc.

Els BSS a més, permeten treballar amb una jerarquia que fa possible definir varis estils per separat, per classes o noms de components. Així un component pot estar definit amb diferents classes apart de la pròpia del component, podent fer que heretin formats i personalitzar detalls segons quina instància d'aquest component sigui.

A més, al tenir en compte també l'herència de les classes, un estili definit a la classe *root* o **BComponent** serà heretat per la resta de components.

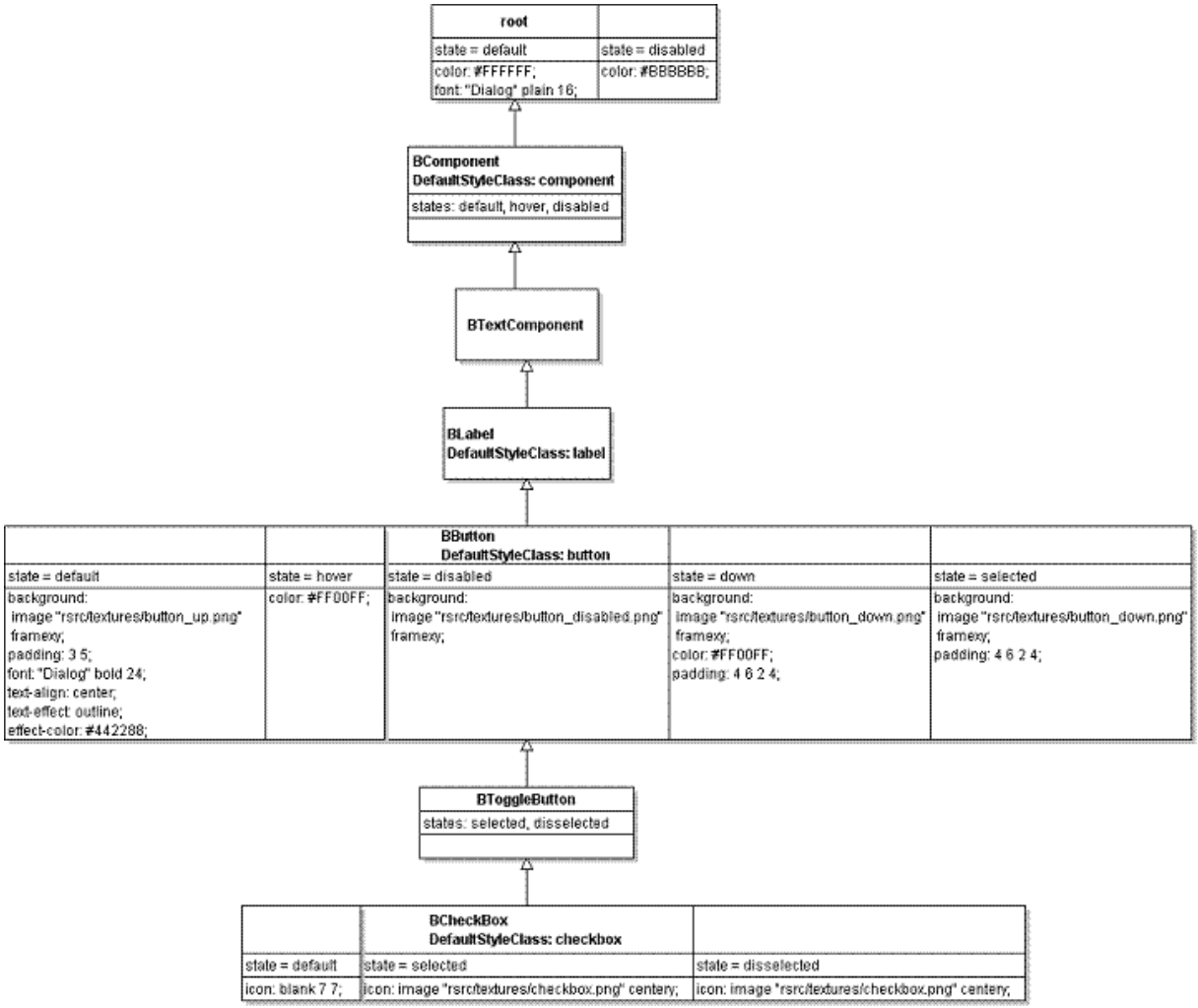


Figura 24: Exemple d'herencia entre elements de GBUI

A la Figura 24 podem veure, per exemple, com els components *BCheckBox*, *BToggleButton*, *BButton*, etc. van heretant dels seus predecessors, a més de, en alguns cassos (*BComponet*, *BLabel*, *BButton* i *BCheckBox*) defineixen el seu propi nom de classe per a donar format.

3.2.4.2. Classes Principals

Les classes principals d'aquest sistema de GUI son **BuiSystem** i **BComponent**. La primera representa la gestió del sistema en si, i només hi interaccionem al iniciar el sistema. La segona representa a tots els components que conformen el sistema de finestres. Totes les classes amb les que tractarem per tal de mostrar text, botons i demés seran herència de **BComponent**.

El número de components que hereten és molt ampli, per tant només tractarem les branques principals. Per una banda, tenim la banda d'herència que composa els botons, texts i demés.

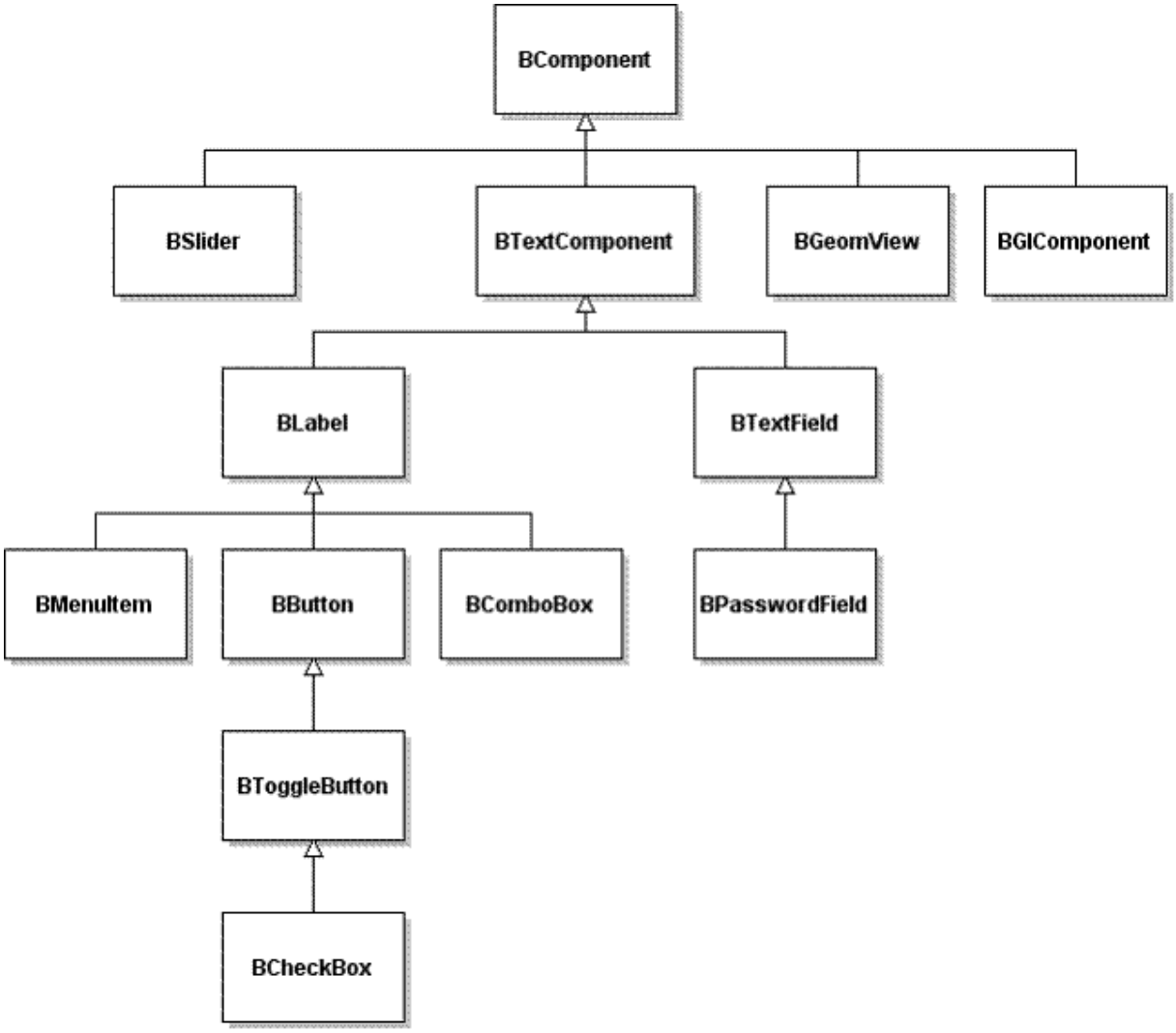


Figura 25: Arbre d'herència parcial de BComponent

En la Figura 25 podem veure part de l'arbre d'herència de la classe **BComponent**, que inclou aquells elements que contenen text i deriven de **BTextComponent**.

Per altra banda, com es pot veure a la Figura 26, hi ha una part de l'herència que representa a les finestres, passant des de l'objecte, també de baix nivell **Bcointainer,** fins a implementar moltes modalitats de finestres, essent la classe **BWindow** la més comú.

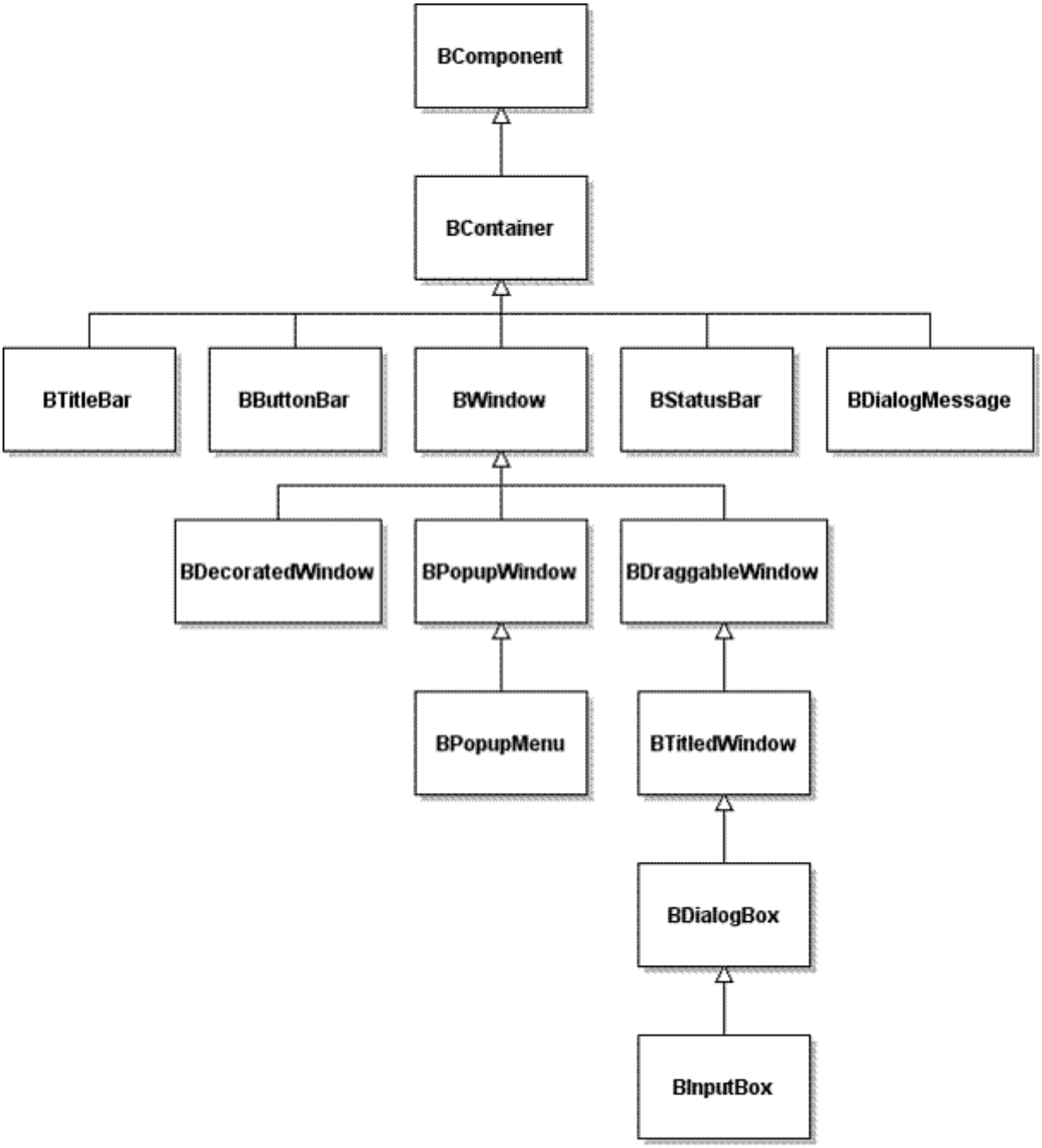


Figura 26: Part de l'abre d'herència de BComponet que mostra les finestres

3.2.5. OpenGL

OpenGL és una especificació estàndard que defineix una API multillenguatge i multiplataforma per a escriure aplicacions que produeïxen gràfics 3D. Va ser desenvolupada originalment per Silicon

Graphics Incorporated (SGI). OpenGL significa Open Graphics Library, que traduït és "biblioteca de gràfics oberta".

Entre les seves característiques podem destacar que és multiplataforma i que la gestió de la generació de gràfics 2D i 3D per maquinari ofereix al programador una API senzilla, estable i compacta. A més la seva escalabilitat ha permès que no s'hagi estancat el seu desenvolupament i també la creació d'extensions, una sèrie d'afegits sobre les funcionalitats bàsiques, per tal d'aprofitar les creixents evolucions tecnològiques.

Per emfatitzar les característiques multillenguatge i multiplataforma de OpenGL, s'han desenvolupat algunes interfícies en molt de llenguatges, en el cas del Java s'ha implementat la LWJGL que és la que fa els accessos a OpenGL així com a OpenAL, entre d'altres.

Pàgina Web: http://www.opengl.org/

3.2.6. OpenAL

OpenAL és una API d'àudio multiplataforma desenvolupada per Creative Labs per el renderitzat eficient de l'àudio posicional i multicanal en tres dimensions. Està pensat per el seu ús en videojocs i l'estil del codi és semblant al de l'OpenGL.

L'API està disponible per a les següents plataformes: Mac OS, Linux (tant per OSS com per ALSA), *BSD, Solaris, Irix, Microsoft Windows, Sony PlayStation 2, Microsoft Xbox y Nintendo GameCube.

L'OpenAL ens proporciona mètodes per tal de poder obtenir el nostre so en tres dimensions. Aquest funcionament es pot dividir en objectes font, receptors i búffers d'àudio. Un objecte font conté un punter a un búffer, a més de una serie d'atributs com la velocitat, la posició, la direcció o la intensitat de l'emissor. Un receptor conté tota la informació dels atributs anteriors a més del guany aplicat a tot el so. Només i pot haver un receptor. Els búffers contenen la informació del so en format PCM. El motor de renderitzat es l'encarregat de controlar tots els càlculs necessaris com l'atenuació, doppler,...

Molts jocs comercials de primera línia utilitzen les llibreries OpenAL. De la llarga llista, podem destacar entre d'altres el Doom 3, Quake 4, Unreal Tournament,...

Pàgina web: http://connect.creativelabs.com/openal/default.aspx

3.2.7. LWJGL

La Lightweight Java Game Library (LWJGL o Biblioteca Java Lleugera per Jocs) és una solució dirigida a programadors tant amateurs com professionals i està destinada a la creació de jocs de qualitat comercial escrits en el llenguatge Java. LWJGL proporciona als desenvolupadors accés a diverses biblioteques multiplataforma, com OpenGL (Open Graphics Library) i OpenAL (Open Audio Library), permetent la creació de jocs d'alta qualitat de gràfics i so 3D. A part, LWJGL també pot accedir a controladors del joc com GamePads, volants i Joysticks.

Totes aquestes funcionalitats estan integrades en una sola API i facilita enormement la creació de videojocs en Java, ja que allibera al programador de fer les complexes crides JNI (Java Native Interface, encarregades de interactuar amb altres programes escrits en altres llenguatges), alhora que proporciona un rendiment espectacular.

Cal remarcar que el LWJGL no és un motor de joc, sinó unes llibreries per donar accés als programadors Java a unes tecnologies que normalment no s'implementen correctament. LWJGL s'ha d'entendre com una API base, sobre la que s'hi recolzen algunes potents eines gràfiques, com és el cas del jMonkeyEngine.

La llicència és BSD, de lliure distribució.

Aquestes llibreries ja estan incorporades a dins del jMonkeyEngine.

Pàgina web: http://lwjgl.org

3.3. Programes utilitzats

3.3.1. Eclipse IDE

L'Eclipse és un entorn de desenvolupament integrat (IDE) de codi obert, multi-plataforma, per a desenvolupar allò que el projecte anomena “Aplicacions de client enriquides”. Aquesta plataforma és molt versàtil, i s'han creat molts plugins per a treballar en ella, i fer-la molt completa.

Eclipse és també una comunitat d'usuaris que amplia constantment les aplicacions que el formen.

Eclipse va ser desenvolupat originalment per IBM com al successor de la seva família d'eines per a VisualAge. Ara està desenvolupat per la Fundació Eclipse, una organització independent sense ànim de lucre que fomenta una comunitat de codi obert i un conjunt de productes complementaris, capacitats i servies.

Eclipse disposa d'un editor de text, amb ressaltat de sintaxis. Compilació en temps real i té proves unitàries amb Junit, control de versions en diferents protocols, suportat gràcies a plugins, com per exemple el SVN que nosaltres hem utilitzat per al nostre projecte. També té suport d'integració amb Ant, assistents per a la creació de projectes, classes, tests, etc.

Eclipse va ser alliberat originalment sota la Common Public License, però després va ser re-llicenciat sota la Eclipse Public License. La Free Software Foundation ha dit que ambdues llicències son de software lliure, però son incompatibles amb la Llicència Pública General de GNU (GNU GPL).

Hem triat aquest IDE per una sèrie de característiques. La primera i més important, ja que en això hem basat el nostre projecte, es que tingués una llicència de software lliure. Eclipse implementa un compilador de Java, necessari per a fer funcionar el motor gràfic utilitzat. També, incorpora un plugin per a gestor del control de versions SVN.

Com a últim detall, amb la mateixa facilitat podíem haver optat per un altre IDE que és NetBeans, ja que dels dos hi havia una bona documentació, però teníem experiència anterior en la utilització d'Eclipse, i això ens ha fet decantar per utilitzar aquest software, ja que així no utilitzàvem gaire temps en l'aprenentatge.

3.3.2. Blender

Blender és un programari lliure multiplataforma, dedicat a l'edició tridimensional. Això inclou eines per al modelatge d'objectes, l'edició de materials, l'animació, el render, compositing amb un sistema de nodes i un motor de jocs, a més de funcionalitats complementàries com l'edició no linial de vídeo.

El programa fou inicialment distribuït de forma gratuïta però sense el codi font. Actualment és compatible amb totes les versions de Microsoft Windows, GNU/Linux, Solaris, FreeBSD, IRIX i Mac OS X.

Originalment, el programa va ser desenvolupat per l'estudi d'animació holandès NeoGeo com una aplicació per a ús intern; més tard el principal autor, Ton Roosendaal, fundà l'empresa "Not a Number Technologies" (NaN) el juny del 1998 per desenvolupar i distribuir el programa. L'empresa va fer fallida el 2002 i els creditors van decidir oferir Blender com a producte de codi font obert i gratuït sota els termes de la GNU GPL. El 18 de juliol del 2003, Roosendaal creà sense ànim de lucre la Fundació Blender per recollir donacions; el codi font es va fer públic el 13 d'octubre.

Hem escollit Blender per les següents característiques:

- Llicència GNU GPL, software lliure com totes les eines triades.
- Multiplataforma, que ens ha permès treballar perfectament sobre GNU/Linux.
- Totes les eines estàndards d'edició de models, malles, textures, etc.
- Incorporació d'ossos, així com càrrega d'influència de les articulacions amb la malla adjunta, a l'hora de provocar deformacions.

- Animació per armadura (ossos).
- Possibilitat d'afegir plugins en llenguatge Python (explicat més endavant, en el software HottBJ).

3.3.3. Audacity

Audacity és un programa multiplataforma d'enregistrament i edició de so. Sota llicència GNU, aquest programa es pot trobar en GNU/Linux, BSD, i altres sistemes operatius com Windows i Mac OS X.

Algunes de les seves principals característiques son:

- Enregistrament d'àudio en temps real.
- Un gran conjunt de plugins que serveixen per a fer-hi efectes digitals. A més, es poden afegir més efectes amb amb el llenguatge Nyquist.
- Edició d'arxius d'àudio dels tipus Ogg Vorbis, MP3, WAV, AIFF, AU i LOF.
- Conversió entre formats d'àudio.
- Importació d'arxius en format MIDI i RAW.
- Edició de pistes múltiples.

3.3.4. Gimp

GIMP (GNU Image Manipulation Program) és, com el seu nom indica, un programa de GNU per al tractament d'imatges. Tot i que originalment les seves sigles significaven General Image Manipulation Program, el 1997 es canviar per el seu nom oficial, ja que és part oficial del projecte GNU. Està creat per voluntaris i distribuït sota la llicència GPL.

Està construït amb les llibreries GTK les quals es van crear pensant en aquest

programa. La primera versió es va desenvolupar en sistemes Unix i va ser pensada especialment per GNU/Linux. No obstant això, actualment existeixen versions totalment funcionals per Windows i Mac OS X.

La biblioteca de controls gràfics GTK desenvolupada per a GIMP va ser l'origen de tot un entorn de finestres: Gnome.

Gimp, per tant, és un programa que serveix per a processar gràfics i fotografies digitals. Tant la creació d'aquests com la modificació, podent fer accions com crear imatges, canviar mides, retallar o modificar les fotografies, modificar els colors, utilització de capes per a combinar imatges, apart d'altres usos més avançats com son les animacions, els vectors o l'edició de vídeo.

Com a curiositat, GIMP es també conegut per ser, segurament, la primera gran aplicació lliure per a usuaris no professionals o experts. És a dir, no és una eina de programadors per a programadors, i per tant es pot considerar una prova o demostració de que el procés de desenvolupament de programari lliure pot crear aplicacions que els usuaris no avançats puguin usar; d'aquesta forma, GIMP va obrir el camí d'altres projectes tant importants com GNOME, KDE, Mozilla Firefox, OpenOffice.org i altres aplicacions posteriors, que hem pogut utilitzar en aquest projecte.

Finalment voldríem comentar que GIMP, en un principi, va ser desenvolupat com una alternativa lliure al Photoshop, però si bé GIMP és un programa molt potent, encara no ha arribat a atrapar el programa d'Adobe, que actualment domina clarament el mercat. Això és degut a varis factors, però cal ser conscient de que en quant a possibilitats per a un dissenyador experimentat Photoshop és més complert, però per els usos que necessitàvem nosaltres, GIMP és molt més del que ens calia.

3.3.5. OpenOffice.org

OpenOffice.org és una suite ofimàtica de programari lliure i codi obert de distribució gratuïta que inclou eines com el processador de texts, full de càlcul, presentacions, eines per al dibuix vectorial i bases de dades. Està

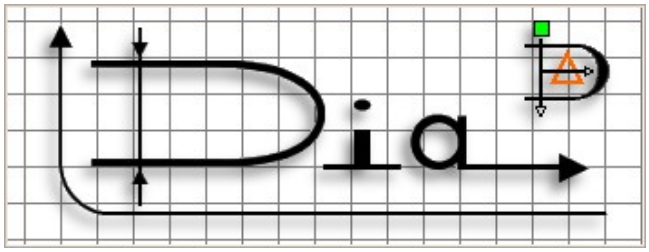
disponible per a moltes plataformes com GNU/Linux, BSD, Solaris, Mac OS i altres Microsoft Windows. OpenOffice.org està creat per a ser altament compatible amb el software privatiu de Mircosoft, Microsoft Office, ja que és el seu principal competidor. A més, suporta l'estàndard ISO OpenDocument, raó per la qual és senzill realitzar un intercanvi de documents amb altres programes.

OpenOffice.org posseeix com a base inicial el StarOffice, una suite ofimàtica que Sun Microsystems (l'empresa desenvolupadora d'OpenOffice.org) va adquirir l'any 1999 i en va alliberar el codi font sota la llicència LGPL el juliol del 2000.

Una curiositat es que tot i que habitualment es denomina de manera informal tant el projecte com l'aplicació en si “OpenOffice”, aquesta és diu “OpenOffice.org” degut a que OpenOffice és una marca registrada que posseeix una altra empresa. Per tant, l'abreviació del nom d'aquesta és OOo, incloent-hi la 'o' del .org.

Nosaltres hem fet servir l'aplicació Writer, que és aquella dedicada a l'edició del text per a editar aquest document, u l'aplicació Impress per a la presentació.

3.3.6. Dia



Dia és una aplicació per a la creació de diagrames tècnics sota llicència GPL basada en les llibreries GTK+. És una potent i còmode eina creada per a GNU/Linux, així com altres sistemes UNIX, i portada també a sistemes Windows.

Entre les seves característiques s'inclouen:

- Suport per a diferents tipus de diagrames com ara diagrames entitat-relació, diagrames de xarxa, diagrames de flux, diagrames d'estructures estàtiques de UML (Unified Modelling Language - diagrames de classe), etc.
- Permet la utilització de formes personalitzades creades per l'usuari com a simples descripcions XML.
- Exportació a múltiples formats (EPS, SVG, CGM, PNG...).

En general, Dia és una utilitat versàtil que pot ser utilitzada per una gran varietat de

camps. Pot ser utilitzada per a totes les possibilitats de l'UML, per un enginyer electrònic per a dissenyar circuits, per un programador per a mostrar el flux d'execució d'un programa, per un administrador de xarxa per a mostrar el disseny de la seva xarxa, com per qualsevol usuari per a fer els seus propis esquemes, ja que inclou una gran varietat de símbols agrupats en múltiples camps, a més dels propis que un mateix es pugui crear.

4. Descripció del videojoc

En aquest apartat descriurem de que va i com haurà de ser el nostre videojoc, així com exposarem a un nivell conceptual i sense entrar en cap moment en la implementació, aquells elements més importants per a la comprensió del joc i del projecte.

4.1. Descripció general

El nostre videojoc pretén ser un senzill però entretingut plataformes de partides ràpides i amb la possibilitat de jugar sol o bé amb algun amic, en partides multijugador.

No seguirà cap fil argumental complex, així que en cada partida que iniciem, l'únic que pretendrem serà aconseguir el màxim de punts (com s'explicarà posteriorment) per a guanyar.

Tot i que en un principi vam pensar en fer un joc amb una gran història èpica de fons, vam veure que seria un projecte inabastable per a desenvolupar en dos persones que estan iniciant-se en aquest món, així que vam optar per un joc senzill però que ens permetés completar-lo.

4.2. Ambientació i argument

Ens trobem a l'any 2291, la Federació Internacional de Carters Units i Simpàtics (FICUS), ha creat el primer torneig mundial de repartidors de paquets, a causa de la creixent admiració cap a l'antic art de repartir paquets.

Tu, un humil carter xinés, has estat seleccionat per la teva habilitat llegendària en tirar paquets a bústies, gossos i vianants, amb un precisió i rapidesa que pocs ulls humans han pogut seguir en lligues menors.

Hauràs de competir pels llocs més recòndits del planeta per tal de demostrar que ets el millor missatger.

4.3. Funcionament del joc

El joc consistirà en fer partides ràpides (d'uns cinc minuts) en les quals cada jugador

haurà de controlar a un personatge que hagi escollit anteriorment, i haurà d'introduir uns paquets en unes determinades bústies (segons una lògica de colors) que prèviament haurà recollit d'unes furgonetes que aniran entrant i sortint de l'escenari.

Cada cop que recollim paquets en una furgoneta, agafarem un nombre concret de paquets (en principi quatre, però podrien ser un nombre variable). Quan un personatge agafi paquets, els “guardarà” en una saca (en sentit metafòric, els personatges no portaran cap saca de correus) i només veurem el paquet que portem a la mà en aquell moment. Un comptador a la pantalla ens indicarà quants paquets més tenim, però no en veurem el color. Quan llenci el que porta a la mà, automàticament se li posarà un nou paquet a la mà, i es restarà un del comptador de paquets. Quan no li quedin paquets, haurà de tornar a anar-ne a buscar a la furgoneta.

En els modes individuals, es competirà per a aconseguir batre les millors puntuacions que s'hagin obtingut en altres partides, mentre que en el mode de varis jugadors, el que s'haurà de fer serà aconseguir derrotar als contrincants.

Per a aconseguir punts, el que haurem de fer serà introduir els paquets dintre de les bústies corresponents, havent de ser aquests del mateix color. A més dels paquets que comparteixin color amb alguna bústia, tindrem paquets d'altres colors que hauran de ser descartats, podent-se fer servir per a atacar als contrincants, com s'explicarà més endavant.

Figura 27: Exemple d'una renderització amb el programa blender d'un pla tal i com volem que quedi col·locada la càmera

L'escenari constarà d'una superfície plana de forma rectangular en el qual veurem tots els costats limitats per elements que faran que el jugador no pugui sortir. A més, fora de l'escenari posarem elements decoratius per a ambientar l'acció.

La vista serà amb una càmera fixa, col·locada a una certa altura, suficient perquè enfoqui el centre de la pantalla, des d'on puguem veure la pantalla sencera, i els jugadors tinguin una mida suficientment gran com a poder-hi jugar. La càmera, per això, no estarà centrada, sinó que estarà fora del límit inferior de la pantalla de manera que veurem els objectes en perspectiva com es pot veure a la Figura 27.

4.3.1. Modalitats

Dintre del funcionament del joc, apart de tenir la opció de jugar amb un sol jugador o amb varis, decidirem si volem jugar o bé contra-rellotge, o bé fins a assolir un determinat nombre de punts.

En la primera opció, guanya aquell que tingui més punts quan el cronometre arribi a zero. En el mode individual, es comparará amb la millor marca registrada.

En la segona opció, tindrem un límit de punts a assolir, i aquell que hi arribi en un temps menor, guanyarà la partida. Com en l'altre mode, si es juga de manera

individual, es tornarà a comptar amb la millor marca registrada d'aquesta modalitat.

4.3.2. Atributs

Un personatge tindrà dos atributs físics bàsics, i una sèrie d'atributs que el definiran com a personatge però no afectaran al joc.

Els atributs físics seran la força i la velocitat. El primer definirà la potencia de tir dels paquets que tindrà el personatge (quan lluny els pot llençar), i el temps que estarà estabornit quan rebi l'impacte d'un paquet (com s'explica més endavant). El segon, la seva velocitat (o agilitat), servirà per a controlar com de ràpid es mou el personatge, així com la velocitat del moviment de llançar un paquet (explicat posteriorment).

Apart, cada personatge tindrà una sèrie d'atributs que el definiran però no afectaran al desenvolupament del joc, com el nom, la seva descripció o quin model 3D farà servir per jugar.

4.3.3. Moviments

El personatge podrà moure's en totes les direccions del pla, a més d'estar quiet i llençar paquets. Tots aquests moviments seran realitzats de manera voluntària, mentre que involuntàriament podrà ser bloquejat per un personatge no jugador (a partir d'ara PNJ) o estabornit per l'impacte d'un paquet.

4.3.3.1. Moviment dels personatges

La relació entre els controls i el moviment serà la següent:

- Si està encarat a la direcció del control que premem, avançarà en aquella direcció de manera normal.
- Si canviem de direcció, girarà de manera gradual mentre es desplaça fins a encarar-se amb aquella direcció que li hem assignat.
- Finalment, si està xocant amb algun element de la pantalla i el fem girar, girarà sobre el propi eix per a poder-se desbloquejar.

4.3.3.2. Llançament de paquets

El moviment de llençar paquets és des del costat oposat al braç que porta el paquet

cap endavant en diagonal. Un cop es deixa anar el paquet, aquest segueix una línia recte respecte a la visió del personatge i fa una paràbola en sentit vertical, fins a arribar al terra.

La distancia a la que arribi el paquet dependrà de la força del personatge (com ja s'ha dit anteriorment). Si el paquet arriba al terra, desapareixerà després d'uns segons. En cas de tocar-lo, perjudicarà al personatge.

Quan es llança un paquet, el personatge es para momentàniament. Romandrà immòbil mentre faci tot el moviment del braç (això fa vulnerable el personatge quan tira paquets a bústies). Com més velocitat tingui, més ràpid llençarà i es podrà tornar a moure.

El personatge farà algun crit o exclamació quan llenci un paquet, per a emfatitzar la importància d'aquest fet, i ja de pas, fer més graciós el joc.

4.3.4. Adversitats

En aquesta secció enumerarem les adversitats amb les que ens podem trobar durant el transcurs de la partida i que ens faran perdre temps per a afavorir als nostres rivals.

4.3.4.1. Rebre l'impacte d'un paquet

Una possible adversitat es produeix al rebre l'impacte d'un paquet, sigui perquè una bústia et retorni un paquet, al no ser aquest del mateix color que la bústia, o sigui perquè un rival et tira un paquet dels que porta.

Al rebre l'impacte d'un paquet, el personatge es desestabilitza, cosa que el fa romandre “quiet” (el personatge es considera bloquejat) durant uns segons, depenent de la força d'aquest personatge.

A més el personatge perdrà dos paquets dels que porti a sobre, si es que en porta. Tot i així, el personatge no perd mai el paquet que du a la mà.

4.3.4.2. Estar retingut per un PNJ

Els personatge no jugadors que es trobin en pantalla, intentaran arribar fins al nostre personatge per tal de retenir-lo amb diferents “excuses”. El temps que estarà retés

dependrà del PNJ i de la velocitat del nostre personatge.

Òbviament, mentre un jugador està retingut, el personatge es considera bloquejat i els altres jugadors poden aprofitar per a llençar-li paquets que entre altres coses li farà perdre els dos paquets. Alhora, si el paquet enlloc de tocar al jugador toca al personatge no jugador, el personatge jugador quedarà alliberat.

4.4. Opcions del videojoc fora de la partida

Amb opcions del videojoc fora de la partida ens referim a totes aquelles coses que podrem fer quan no estiguem jugant, sigui per configuració, opcions de cara a iniciar partida, o pantalles de consultar dades, crèdits, etc.

Per iniciar una partida, tindrem dos maneres separades. Podrem iniciar partides de manera individual o multijugador. En aquests casos, que son similars, hauríem d'escollir el jugador o jugadors, en cas de multijugador, que farem servir, el tipus de joc i la pantalla.

Apart, hem de poder configurar les opcions més bàsiques del joc, com podrien ser la resolució o l'activació/desactivació del so. Una altra opció més important que caldria també tenir en compte, seria una petita configuració de controls, ja que al haver-hi un multijugador de fins a quatre persones, necessitarem configurar controladors de joc addicionals al nostre teclat per a poder-hi jugar.

Finalment, dos pantalles amb les quals el jugador no interacciona, per a consultar les millors puntuacions i els crèdits de l'aplicació. En aquestes pantalles no s'implementarà cap tipus de funcionalitat, només la de poder tornar enrera.

5. Requeriments del Sistema

En aquest capítol seran descrits els requeriments, els quals recullen, a grans trets, els objectius de l'aplicació juntament amb les seves funcionalitats desitjades. En aquest projecte s'ha utilitzat una metodologia Orientada a Objectes basada en UML pel desenvolupament del software. La nomenclatura UML no defineix una metodologia concreta de desenvolupament de programari Orientat a Objectes, sinó que és un llenguatge que permet modelar, construir i documentar els elements que conformen un sistema Orientat a Objectes.

Dins d'una aplicació apareixen bàsicament dos tipus de requeriments:

- **Requeriments funcionals:** Descriuen quins són els serveis que ens oferirà l’aplicació independentment de la implementació.
- **Requeriments no funcionals:** Ens informen sobre les restriccions que venen imposades pel client o pel propi problema.

A continuació descriurem en detall els dos tipus de requeriments esmentats.

5.1. Requeriments Funcionals

En aquest apartat descriurem els serveis que oferirà aquesta aplicació, sense tenir en compte la seva implementació.

5.1.1. Requeriments generals del sistema

Entenem com a requeriments generals del sistema les principals funcionalitats del programari a desenvolupar, reflectint en tot moment les responsabilitats del programa construït. En aquest apartat posem aquelles necessitats a satisfer en termes globals, sense entrar en detall de com es resoldran.

A continuació es mostren els principals requeriments:

- Iniciar partida d'un sol jugador.
- Poder iniciar també una partida per a varis jugadors simultanis, contemplat com una opció diferent a l'anterior.

- Poder gestionar la configuració de l'aplicació.
- Visualitzar una pantalla estàtica amb un registre de màximes puntuacions en el mode individual.
- Una pantalla on poder visualitzar els crèdits relacionats amb els creadors i col·laboradors de l'aplicació.
- Una opció per a tancar el programa netament, tancant primer tots els subprocessos iniciats.

5.1.2. Identificació dels actors

A continuació cal identificar els actors de l’aplicació. Un actor és una entitat externa (persona, sistema, subsistema...) que interactua amb el sistema interpretant un determinat rol o un determinat estat.

A la nostra aplicació no hi ha cap mena de manteniment ja que es tracta d’un programa en el qual no apareix distinció entre els possibles usuaris que el puguin utilitzar. Per tant, podem concloure l’existència d’un únic actor, l’usuari que interactua en tot moment amb el sistema. Tot i així, aquest usuari serà representat en alguns moments per varis actors ja que podrà agafar diferents estats.

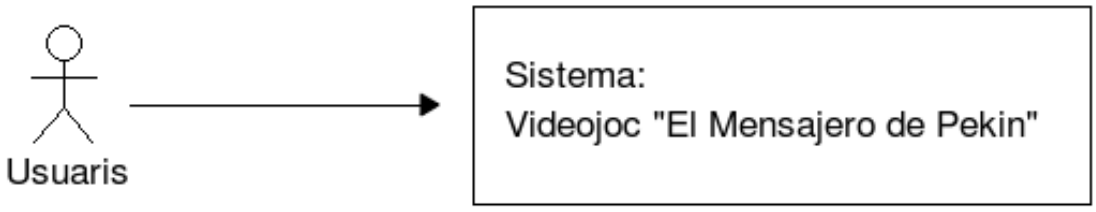


Figura 28: Identificació dels actors

Com es pot veure en la Figura 28, l’esquema és força senzill ja que tots els usuaris que utilitzin l’aplicació disposen dels mateixos privilegis.

5.1.3. Diagrames de casos d'ús generals

Els casos d'ús descriuen el comportament d'un sistema des del punt de vista de l'usuari. Permeten definir els límits del sistema i les relacions entre aquest i el seu entorn. Podem dir que són descripcions de les funcionalitats del sistema independentment de la implementació. Estan basats en el llenguatge natural, de manera que puguin ser accessibles per tots els usuaris.

A continuació, explicarem els diagrames de casos d'ús generals separant el que és iniciar el joc i moure's per els menús, amb el que és el desenvolupament de la partida.

Les funcionalitats exposades en els següents diagrames de casos d'ús (figures X i Y) s'han intentat compactar i simplificar per facilitar-ne la seva comprensió tot i les múltiples relacions que existeixen entre elles. Al següent capítol (anàlisi del sistema), s'exposaran algunes de les funcionalitats més complexes de forma més detallada.

5.1.3.1. Diagrama de casos d'ús: Menú principal

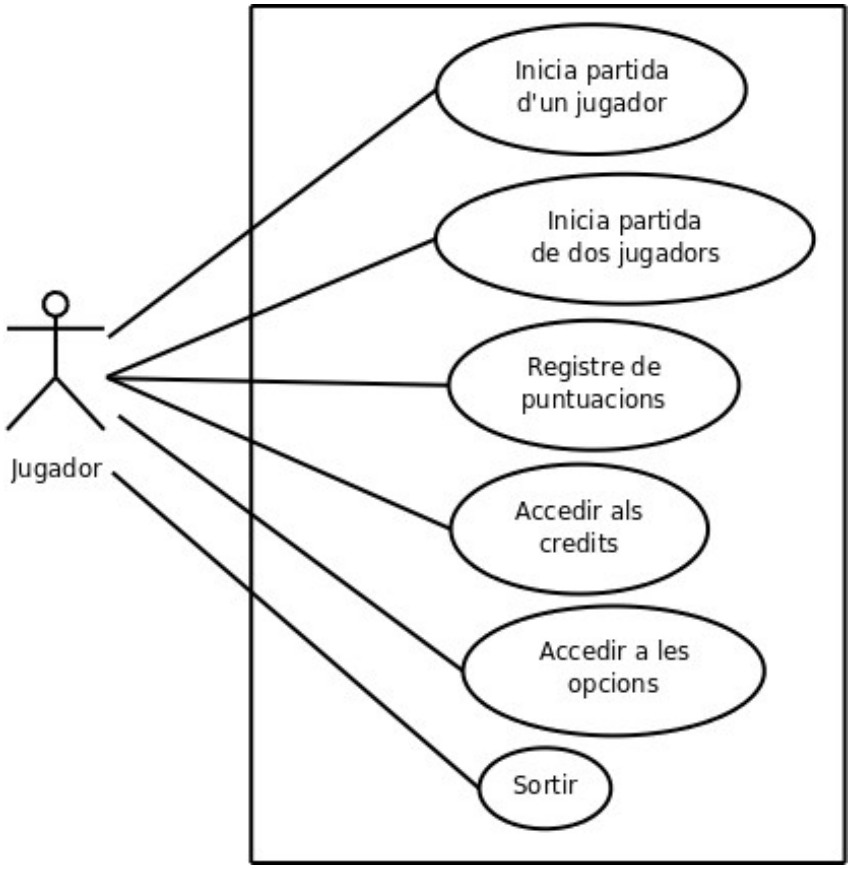


Figura 29: Diagrama de cas d'ús: Menú principal

En la Figura 29 podem veure el cas d'ús general que es produeix quan l'usuari engega l'aplicació. Les funcionalitats que hi surten, corresponen a totes les opcions del menú principal.

Aquests casos d'ús permeten una interacció bàsica amb l'aplicació. Així doncs, un usuari pot iniciar una partida en mode individual, en mode de varis jugadors, pot accedir a les opcions de configuració, o bé visualitzar les pantalles no interactives de crèdits, registre de puntuacions o sortir de l'aplicació.

5.1.3.2. Diagrama de casos d'ús: Partida

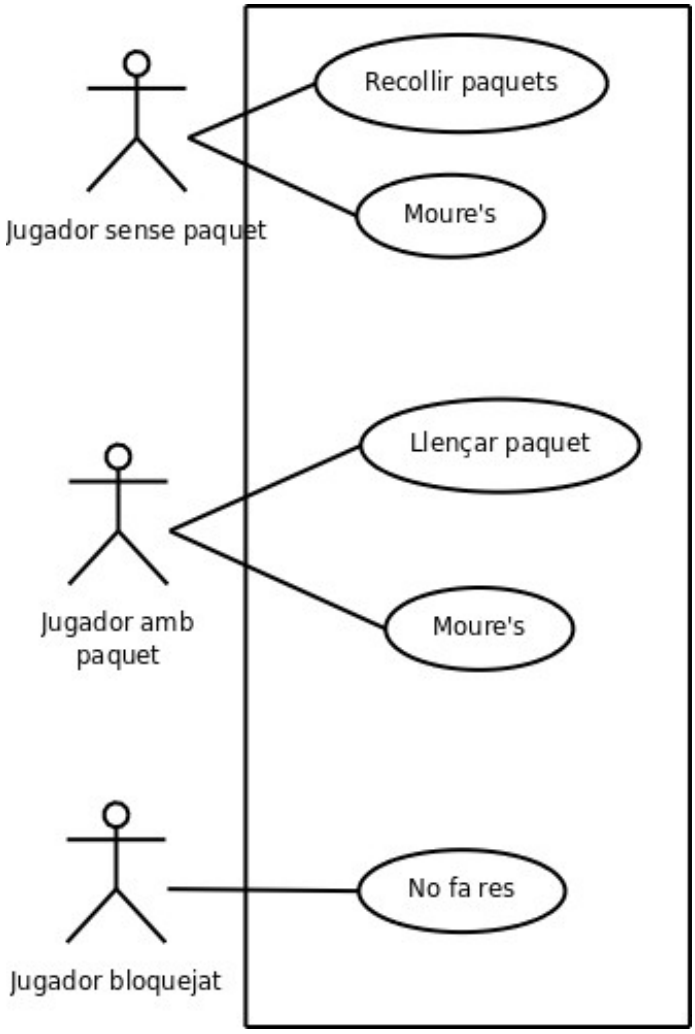


Figura 30: Diagrama de casos d'ús: Partida

En la Figura 30 podem veure el cas d'ús general que es produeix al iniciar pròpiament una partida del joc. Podem veure que hi han tres actors, que representen els tres estas en que un mateix jugador es pot trobar.

Aquest cas d'ús representa una interacció bàsica entre el jugador i el sistema o altres jugadors, i les accions que aquest pot dur a terme, representades per les funcionalitats, o bé els estats en que es pot trobar el jugador, que com hem dit abans, estan representats pels diferents actors del cas d'ús.

5.1.4. Detalls dels requeriments del sistema

Com que arribats a aquest punt ja tenim els primers diagrames de cas d'ús, però aquests son molt senzills ja que és una visió molt global de l'aplicació, esperarem a fer la majoria de les fitxes de cas d'ús als diagrames refinats. En aquest apartat, només posarem aquelles fitxes que no es vegin ampliades al fer-ne un refinament.

5.1.4.1. Cas d'us: Mostrar crèdits

Cas d'ús: MOSTRAR CRÈDITS	
Descripció	Carreguem totes les configuracions del sistema, i accedint des del menú principal als crèdits, els mostrem per pantalla.
Actor	Jugador
Precondició	Cap.
Flux principal	1 Iniciar sistema. 2 Llegir el fitxer de configuració. 2.1 Control d'errors. 2.2 Carregar configuració per defecte. 3 Mostrar menú principal. 4 Accedir a l'opció *Mostrar Crèdits*.
Flux alternatiu	Cap.
Postcondició	Mostrar per pantalla els crèdits del videojoc.
Observacions	Des de la pàgina de crèdits tant sols podrem tornar al menú principal.

5.1.4.2. Cas d'ús: Mostrar màximes puntuacions

Cas d'ús: MOSTRAR MÀXIMES PUNTUACIONS	
Descripció	Carreguem totes les configuracions del sistema, i accedint des del menú principal al les màximes puntuacions, les mostrem per pantalla.
Actor	Jugador
Precondició	Cap.

Flux principal	1 Iniciar sistema. 2 Llegir el fitxer de configuració. 2.1 Control d'errors. 2.2 Carregar configuració per defecte. 3 Mostrar menú principal. 4 Accedir a l'opció Màximes *puntuacions*. 5 Carregar fitxer amb les màximes puntuacions. 5.1 Control d'errors. 6 Mostra formatades les màximes puntuacions.
Flux alternatiu	Cap.
Postcondició	Mostrar per pantalla els crèdits del videojoc.
Observacions	Des de la pàgina de màximes puntuacions només podrem tornar al menú principal.

5.1.4.3. Cas d'ús: Moure's (sense paquets)

Cas d'ús: MOURE'S	
Descripció	Moure's lliurement per la pantalla sense paquets.
Actor	Jugador.
Precondició	Ens trobem dins de la partida, l'usuari no està bloquejat, ni té paquets, fet que faria canviar d'actor.
Flux principal	1 Ens movem amb el control configurat.
Flux alternatiu	Cap.
Postcondició	Moure't dins de la partida.
Observacions	En el cas de col·lisió amb un paquet o un personatge no jugador, ens quedarem bloquejats.

5.1.4.4. Cas d'ús: Recollir paquets (jugador sense paquets)

Cas d'ús: RECOLLIR PAQUETS	
Descripció	Desplaçar-se fins a la furgoneta per tal de recollir paquets.
Actor	Jugador.
Precondició	Ens trobem dins de la partida, l'usuari no està bloquejat, ni té paquets.
Flux principal	1 Ens movem amb el control configurat. 2 Anem direcció cap a una furgoneta. 3 Quan col·lisionem ens bloquegem. 4 Ens posem en estat immune. 5 Recollim paquets. 6 Deixem d'estar immunes.
Flux alternatiu	Cap.

Postcondició	Aconseguir que el jugador tingui paquets.
Observacions	En estat immune, no ens afecten els paquets, i els personatges no jugadors no ens persegueixen.

5.1.4.5. Cas d'ús: Moure's (jugador amb paquets)

Cas d'ús: MOURE'S	
Descripció	Moure's lliurement per la pantalla amb paquets.
Actor	Jugador amb paquets.
Precondició	Ens trobem dins de la partida, l'usuari no està bloquejat, i té paquets.
Flux principal	1 Ens movem amb el control configurat.
Flux alternatiu	Cap.
Postcondició	Moure't dins de la partida.
Observacions	• En el cas de col·lisió amb un paquet o un personatge no jugador, ens quedarem bloquejats. • Mentre tinguem paquets, el jugador podrà llançar-los lliurement. Quan es quedi sense paquets, passa a ser un actor sense paquets.

5.2. Requeriments no funcionals

A l'hora de dur a terme qualsevol mena de projecte és necessari prestar especial atenció a tots aquells aspectes que han de ser tinguts en compte a l'hora de dissenyar el sistema, més enllà de l'explicació funcional detallada anteriorment. Els requeriments no funcionals del sistema són aquells que fan referència a restriccions del tipus: disponibilitat de recursos, seguretat, interfícies externes (hardware i software), entre d’altres. Aquestes condicions ens permetran executar l’aplicació sense cap problema.

Pel que fa a l'aplicació implementada cal dir que des del punt de vista de seguretat, no és requerit cap tipus de control d'accés al programari, ja que es tracta d'una aplicació on els usuaris que hi accedeixin només tindran un rol. A més, no es disposa de dades confidencials ni d'alt risc, per tant no tindria sentit un sistema de protecció de dades.

D'altra banda, cal dir que l'aplicació ha estat implementada sobre una plataforma GNU/Linux, tot i que les eines utilitzades en el desenvolupament es podria haver triat

qualsevol altre plataforma, nosaltres em escollit GNU/Linux per les següents raons:

- La principal raó per escollir GNU/Linux, deixant de banda les raons tècniques i econòmiques, es que és software lliure.
- Ofereix, en determinades ocasions, major fiabilitat i estabilitat.
- Una gran flexibilitat de configuració, suport de diversos tipus de hardware i interoperativitat amb altres sistemes..
- Totes les eines utilitzades en el projecte són de codi obert, amb llicències de software lliure.
- Finalment, dir que la utilització d'aquest sistema al llarg de tota la carrera ha fet que el coneixement sobre aquest sigui major.

La implementació i les proves de l'aplicació s'han dut a terme sobre la distribució Ubuntu 8.04 i posteriorment amb la 9.04. La configuració dels diversos equips en que s'han dut les proves son:

Dell XPS M1330

Intel Core 2 Duo T8100 2.10Ghz
NVIDIA GForce 8400 GS
3Gb de memòria RAM

Dell XPS M1330

Intel Core 2 Duo T7500 2.2Ghz
NVIDIA GForce 8400 GS
3Gb de memòria RAM

L'aplicació està implementada en Java i s'executa mitjançant el Java WebStart característica que la fa multiplataforma. Així que es pot executar des de qualsevol sistema operatiu, tan GNU/Linux, com en Windows, o MAC OS X, l'únic necessari és tenir instal·lat el Java, ja que des de la versió 1.4 el Java WebStart ja ve per defecte.

Pel que fa als dispositius d'entrada, l'usuari pot interactuar amb el sistema mitjançant el ratolí i el teclat o un control de jocs (GamePad).

6. Anàlisi del sistema

En aquest apartat el que volem és aprofundir en els requeriments generals del sistema, exposats anteriorment. El principal objectiu d'aquesta etapa d'anàlisi és obtenir una comprensió precisa de les necessitats del sistema, és a dir, s'encarrega de la investigació a fons del problema a resoldre (que), sense preocupar-nos de trobar una solució (com) d'aquest problema. En el transcurs d'aquest procés, es traduiran les necessitats comentades en el capítol de requeriments a un llenguatge més formal a través de la Enginyeria del Software.

En els models que es definiran tot seguit es centraran en les vistes d'usuari, com per exemple els diagrames de casos d'ús i en les vistes estructurals, com serien els diagrames d'activitats.

Tots els diagrames creats en les següents subseccions han estat creats amb el programa de software lliure Dia, explicat en la corresponent secció.

En aquesta memòria es persegueixen dos objectius, per una banda mostrar les etapes complertes durant el desenvolupament del projecte per tal de donar a conèixer els elements teòrics utilitzats. D'altra banda es vol documentar el sistema final. Per aquest motiu, per exemple, en aquesta etapa d'anàlisi presentem els diagrames en l'etapa final, és a dir, incorporant decisions de disseny. Aquest fet ve donat perquè el desenvolupament orientat a objectes permet que els mateixos models siguin utilitzats de manera iterativa en les diferents etapes de la vida del programari, fent-los créixer des dels requeriments fins a la implementació final.

6.1. Casos d'ús refinats

A partir dels diagrames de casos d'ús generals *Menú principal* i *Partida* representats en el capítol anterior, veiem que la majoria de casos d'ús haurien de ser estudiats amb més detall per tal d'entendre més correctament el seu funcionament.

Tot seguit refarem amb major nivell de refinament aquells que considerem que ho necessiten a causa de ser més importants i/o més complexes.

6.1.1. Casos d'ús del Menú principal

Aquí descriurem els casos d'ús refinats corresponents al cas d'ús general *Menú principal*.

6.1.1.1. Cas d'ús: Inicia partida d'un jugador

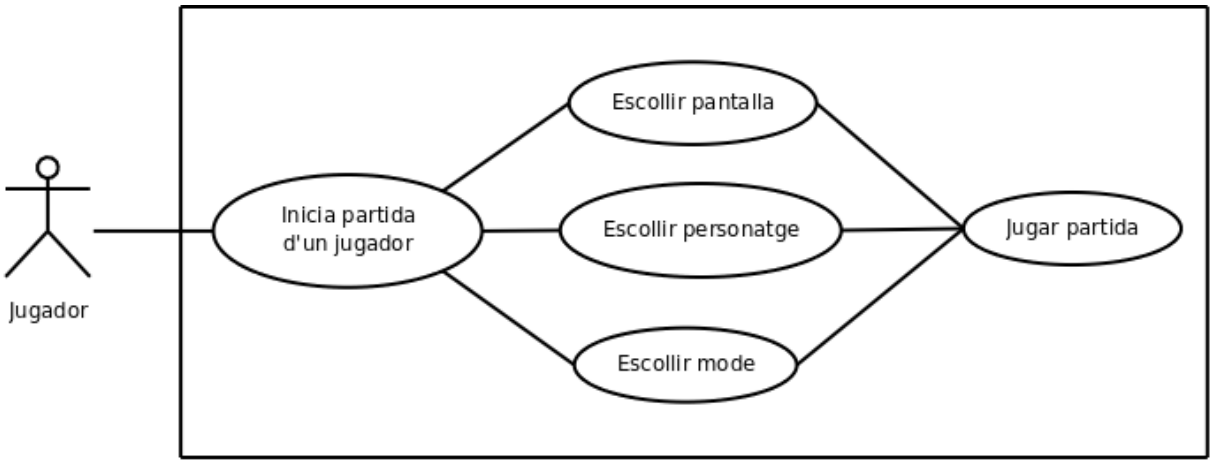


Figura 31: Cas d'ús: Inicia partida d'un jugador.

6.1.1.1.1. Fitxa del cas d'ús

Cas d'ús: INICIAR PARTIDA D'UN JUGADOR	
Descripció	Accedirem al menú de partida individual, després escollirem les opcions proposades i finalment iniciarem una partida.
Actor	Jugador
Precondició	Cap.
Flux principal	1 Iniciar sistema. 2 Llegir el fitxer de configuració. 2.1 Control d'errors. 2.2 Carregar configuració per defecte. 3 Mostrar menú principal. 4 Accedir a l'opció *Iniciar partida individual*. 4.1 Carregar fitxer on es troben les rutes i noms de les imatges. 4.2 Control d'errors. 4.3 Carregar les imatges dels escenaris. 4.4 Carregar les imatges dels jugadors. 5 Mostrar el menú d'iniciar partida d'un jugador. 5.1 Escollir pantalla entre les proposades. 5.2 Escollir personatge entre els possibles. 5.3 Escollir mode de joc, per temps o bé per número

	de punts. 6 Iniciar partida.
Flux alternatiu	En el punt 6, el jugador podrà tornar al menú principal.
Postcondició	Partida iniciada correctament i jugador controla el personatge.
Observacions	En el punt 6 es crearà una nova partida tenint en compte els paràmetres que seran passats des del menú específic d'iniciar partida d'un jugador.

6.1.1.2. Cas d'ús: Inicia partida multijugador

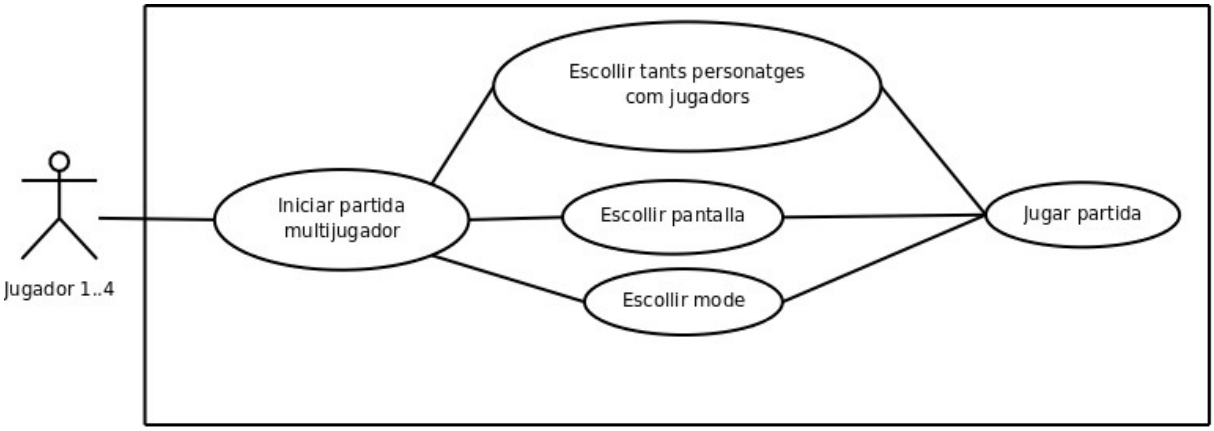


Figura 32: Cas d'ús: Inicia partida multijugador

6.1.1.2.1. Fitxa del cas d'ús

Cas d'ús: INICIAR PARTIDA MULTIJUGADOR	
Descripció	Accedirem al menú de partida per a varis jugadors, després escollirem les opcions proposades i finalment iniciarem una partida.
Actor	Jugador 1..4
Precondició	Per a cada jugador per sobre del segon, necessitem un controlador de joc (game pad).
Flux principal	1 Iniciar sistema. 2 Llegir el fitxer de configuració. 2.1 Control d'errors. 2.2 Carregar configuració per defecte. 3 Mostrar menú principal. 4 Accedir a l'opció *Iniciar partida multijugador*. 4.1 Carregar fitxer on es troben les rutes i noms de les imatges. 4.2 Control d'errors. 4.3 Carregar les imatges dels escenaris. 4.4 Carregar les imatges dels jugadors.

	5 Mostrar el menú d'iniciar partida multijugador. 5.1 Escollir pantalla entre les proposades. 5.2 Per a cada jugador 5.2.1 Escollir personatge entre els possibles. 5.3 Escollir mode de joc. 6 Iniciar partida.
Flux alternatiu	En el punt 6 el jugador podrà tornar al menú principal.
Postcondició	Partida iniciada correctament i els jugadors controlen els personatges.
Observacions	• En el punt 6 es crearà una nova partida tenint en compte els paràmetres que seran passats des del menú específic d'iniciar partida multijugador. • Els personatges seran controlats segons els controladors definits al fitxer de configuració per defecte o modificat en el menú d'opcions.

6.1.1.3. Cas d'ús: Accedir a les opcions

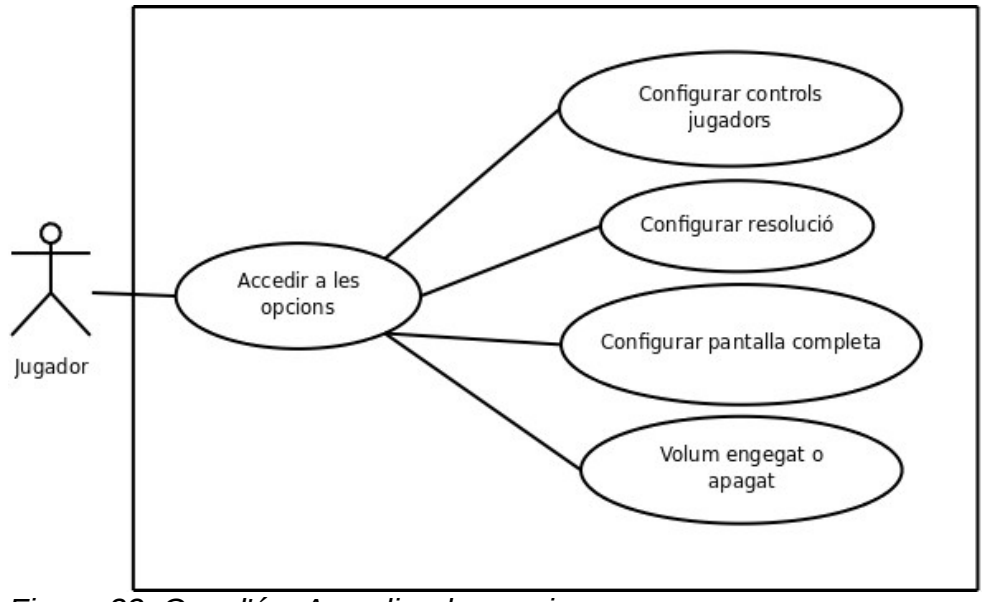


Figura 33: Cas d'ús: Accedir a les opcions

6.1.1.3.1. Fitxa del cas d'ús

Cas d'ús: ACCEDIR A LES OPCIONS	
Descripció	Accedim a la pantalla de configuració, on podrem modificar unes opcions que seran guardades per seguir fent servir un cop reiniciada l'aplicació.
Actor	Jugador

Precondició	Cap.
Flux principal	1 Iniciar sistema. 2 Llegir el fitxer de configuració. 2.1 Control d'errors. 2.2 Carregar configuració per defecte. 3 Mostrar menú principal. 4 Accedir a l'opció del menú *Opcions*. 4.1 Carrega la configuració per defecte. 5 Per cada número de jugador. 5.1 Escollir tipus de controlador. 6 Escollir resolució. 7 Escollir l'activació permanent de la pantalla completa. 8 Habilitar o deshabilitar el so. 9 Guardar les dades.
Flux alternatiu	9 El jugador tindrà la possibilitat de tornar al menú principal.
Postcondició	Ens trobem al menú principal amb les noves dades guardades correctament i carregades al sistema.
Observacions	En el punt 4.1 fem servir les dades de configuració per defecte llegides en el punt 2.2.

6.1.2. Casos d'ús de partida

En aquest apartat descriurem aquells casos d'ús refinats que deriven del cas d'ús general Partida.

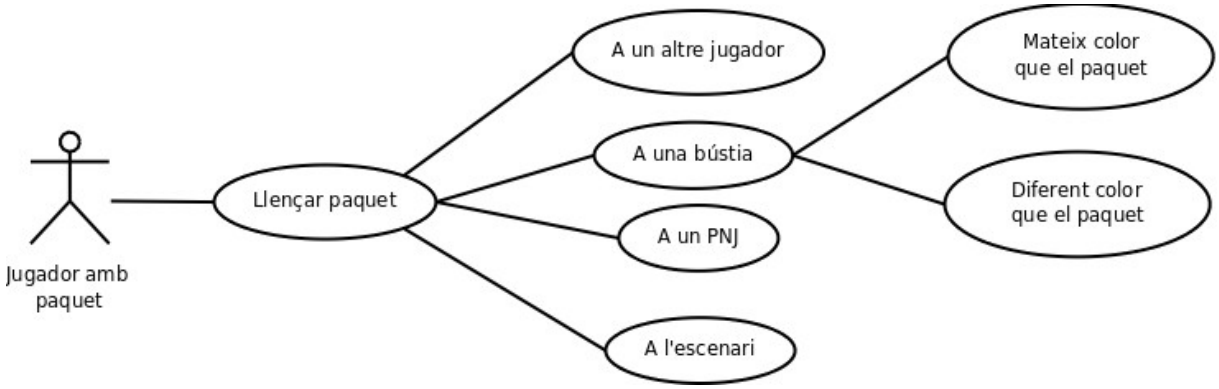


Figura 34: Casos d'ús de partida

6.1.2.1.1. Fitxa del cas d'ús: Llençar paquet a un altre jugador

Cas d'ús: LLENÇAR PAQUET A UN ALTRE JUGADOR	
Descripció	Un jugador amb paquets fa l'acció de llençar un paquet contra un altre jugador i l'abat.
Actor	Jugador amb paquets
Precondició	La partida inicialitzada.

Flux principal	1 El jugador es mou en direcció a un altre jugador. 2 Llença el paquet 3 Encerta a l'altre jugador i el deixa bloquejat.
Flux alternatiu	3. Si no encerta el llençament, passa a ser el cas d'ús: Llençar paquet a l'escenari.
Postcondició	El jugador amb un paquet menys, i l'altre jugador bloquejat.
Observacions	Si el paquet llençat és el darrer, el jugador passa a l'estat, jugador sense paquets.

6.1.2.1.2. Fitxa del cas d'ús: Llençar paquet a una bústia

Cas d'ús: LLENÇAR PAQUET A UNA BÚSTIA	
Descripció	Un jugador amb paquets fa l'acció de llençar un paquet contra una bústia.
Actor	Jugador amb paquets
Precondició	Partida inicialitzada.
Flux principal	1 El jugador es mou en direcció a una bústia. 2 Llença el paquet. 3 Encerta a la bústia. **4 Si encerta bústia del mateix color que el paquet.** 4.1 Suma un punt al registre del jugador. **5 Altrament.** 5.1 La bústia tira el paquet en la mateixa direcció en la que la rebut.
Flux alternatiu	3. Si no encerta el llançament, passa a ser el cas d'ús: Llençar paquet a l'escenari.
Postcondició	El jugador amb un paquet menys.
Observacions	• Si el paquet llençat és el darrer, el jugador passa a l'estat, jugador sense paquets. • 5.1 Si el paquet tirat per la bústia toca el jugador, aquest es queda bloquejat durant uns segons.

6.1.2.1.3. Fitxa del cas d'ús: Llençar paquet a un personatge no jugador

Cas d'ús: LLENÇAR PAQUET A UN PERSONATGE NO JUGADOR	
Descripció	Un jugador amb paquets fa l'acció de llençar un paquet contra un personatge no jugador.
Actor	Jugador amb paquets
Precondició	Partida inicialitzada.
Flux principal	1 El jugador es mou en direcció al personatge no jugador. 2 Llença el paquet. 3 Encerta el tir i toca al personatge no jugador.

	3.1 Aquest personatge no jugador resta bloquejat durant uns segons.
Flux alternatiu	3. Si no encerta el llançament, passa a ser el cas d'ús: Llençar paquet a l'escenari.
Postcondició	El jugador amb un paquet menys i el personatge no jugador quedarà bloquejat durant uns segons.
Observacions	Si el paquet llençat és el darrer, el jugador passa a l'estat, jugador sense paquets.

6.1.2.1.4. Fitxa del cas d'ús: Llençar paquet a l'escenari

Cas d'ús: LLENÇAR PAQUET A L'ESCENARI	
Descripció	Un jugador amb paquets fa l'acció de llençar un paquet contra l'escenari.
Actor	Jugador amb paquets
Precondició	Partida inicialitzada.
Flux principal	1 El jugador es mou lliurement. 2 Llença el paquet 3 Aquest paquet llençat xoca amb l'escenari. 3.1 El paquet s'elimina.
Flux alternatiu	3. Si en el camí, el paquet xoca contra una bústia, un personatge,... ens trobarem en un dels casos d'ús anteriors.
Postcondició	El jugador amb un paquet menys.
Observacions	• Si el paquet llençat és el darrer, el jugador passa a l'estat, jugador sense paquets. • En general, qualsevol paquet llençat que no xoqui amb cap element caurà a l'escenari.

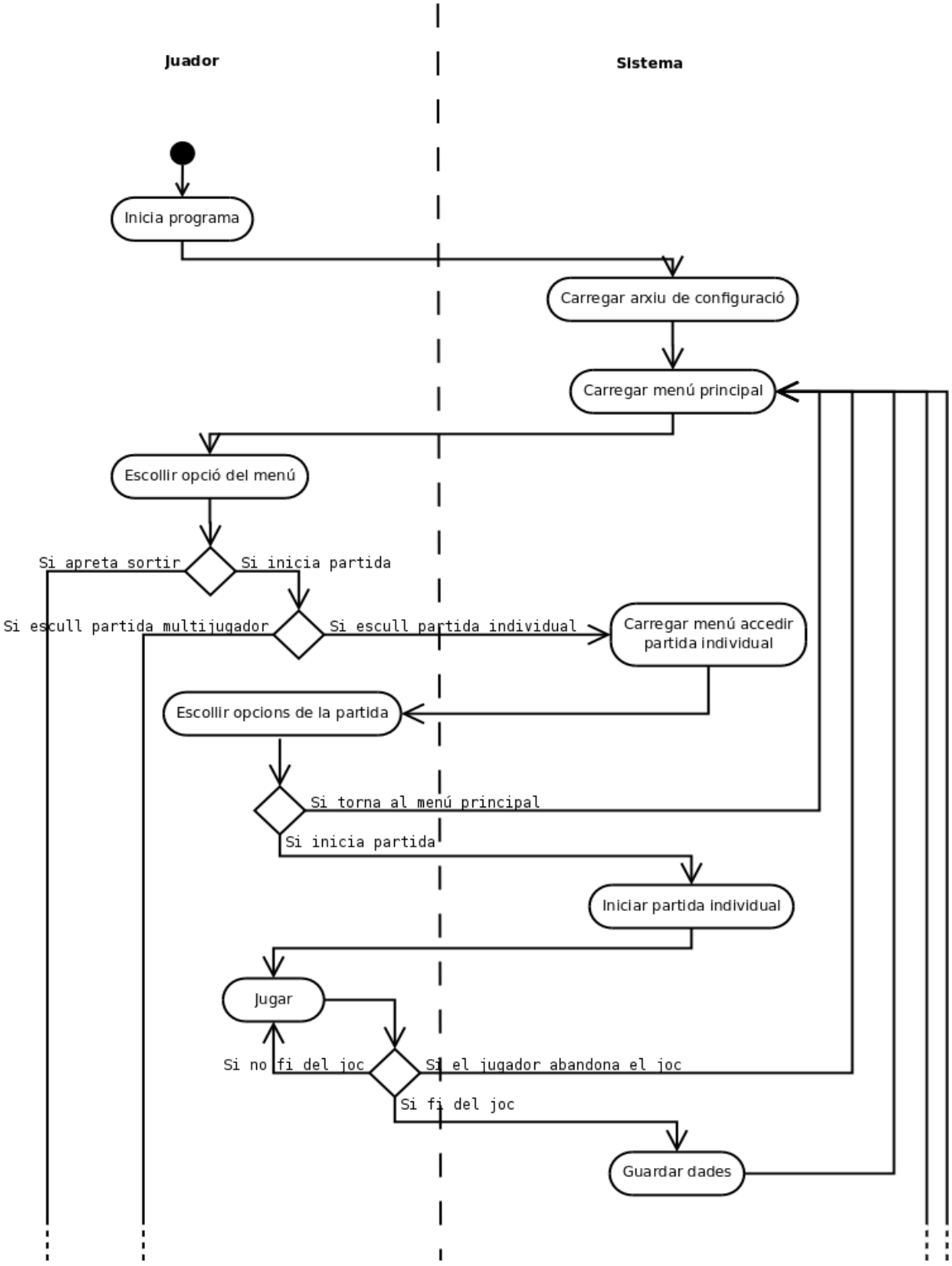
6.2. Diagrames d'activitat

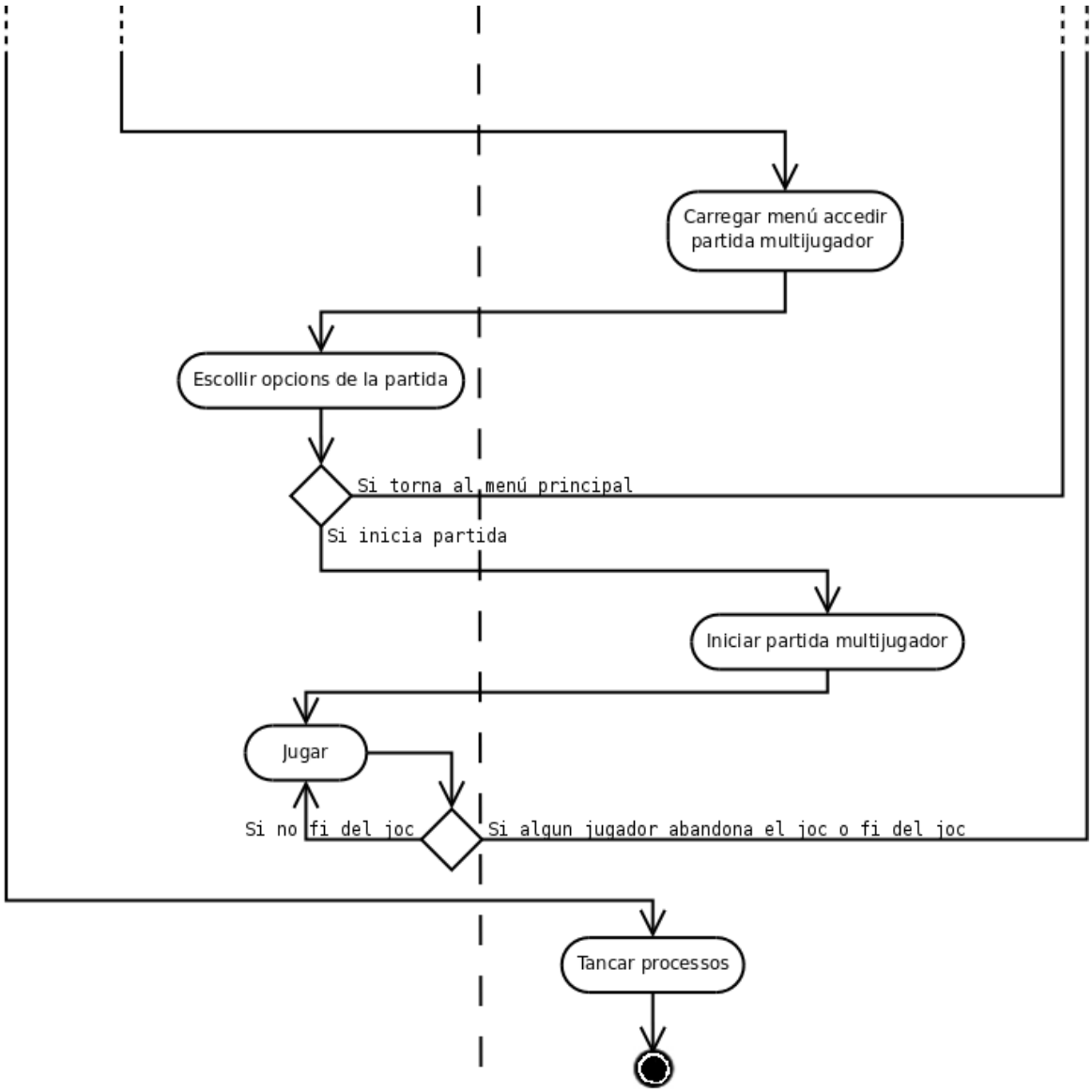
Els diagrames d'activitat mostren fluxos de dades d'activitats i accions, amb suport per a decisions, iteracions i concurrència. En UML, els diagrames d'activitat es poden fer servir per a descriure el negoci i fluxos de dades pas a pas en un sistema. Els diagrames d'activitat no proporcionen informació del comportament d'un objecte o de les col·laboracions entre ells.

6.2.1. Diagrama d'activitat: Iniciar partida

En aquest diagrama es vol representar com serà el flux de la informació tant si es vol iniciar una partida individual o una partida multijugador.

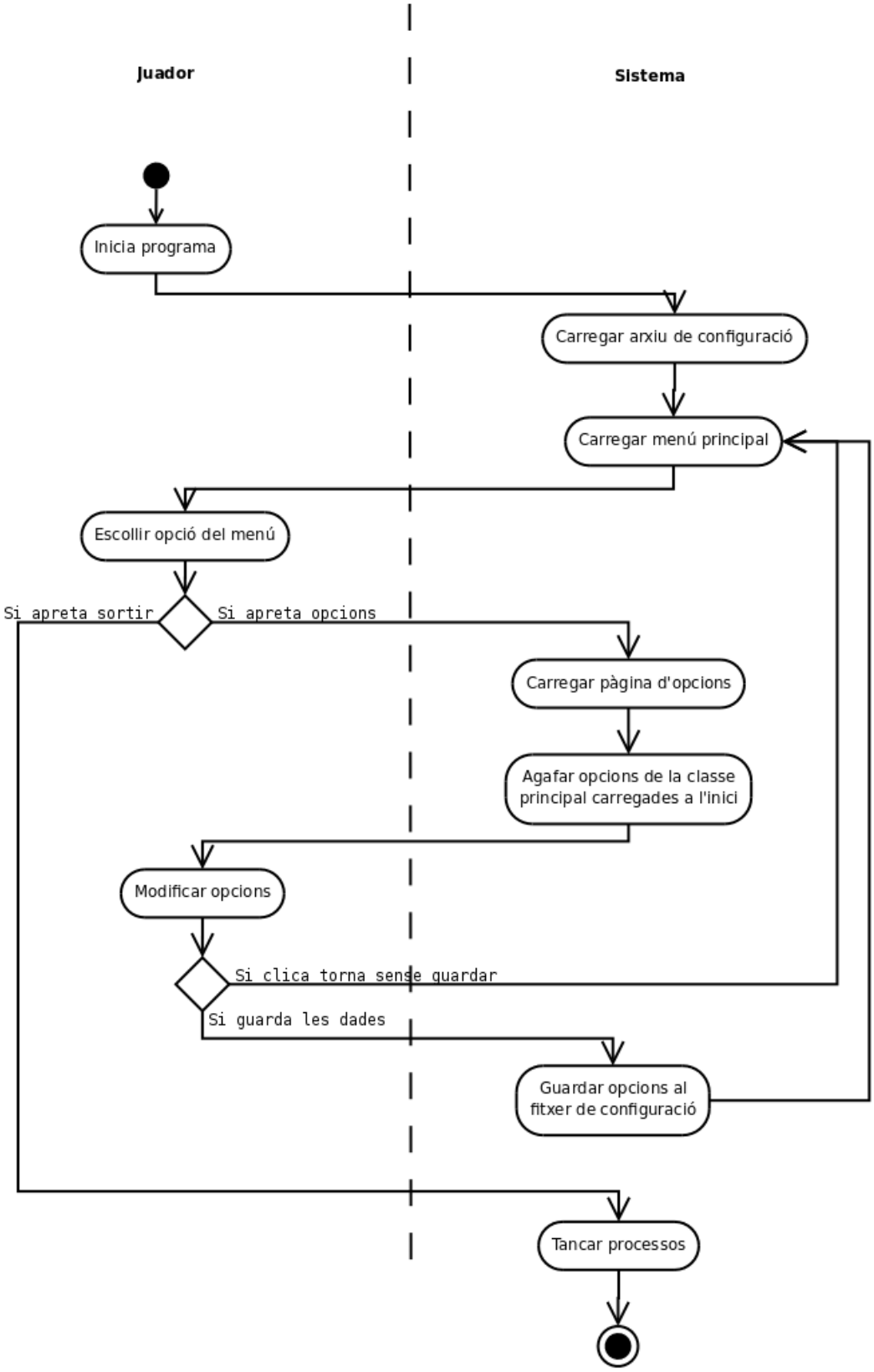
Primer de tot iniciem el programa i el sistema s'ocupa de carregar la configuració i el menú principal. Des d'aquí podem accedir a la partida individual o multijugador. Un cop en qualsevol d'aquests dos casos, escollirem l'escenari, el o els personatges i el mode, i jugarem fins que avortem la partida o finalitzi. En qualsevol dels casos, tornarem al menú principal.


Juador
Sistema
Inicia programa
Carregar arxiu de configuració
Carregar menú principal
Escollir opció del menú
Si apreta sortir
Si inicia partida
Si escull partida multijugador
Si escull partida individual
Carregar menú accedir partida individual
Escollir opcions de la partida
Si torna al menú principal
Si inicia partida
Iniciar partida individual
Jugar
Si no fi del joc
Si el jugador abandona el joc
Si fi del joc
Guardar dades

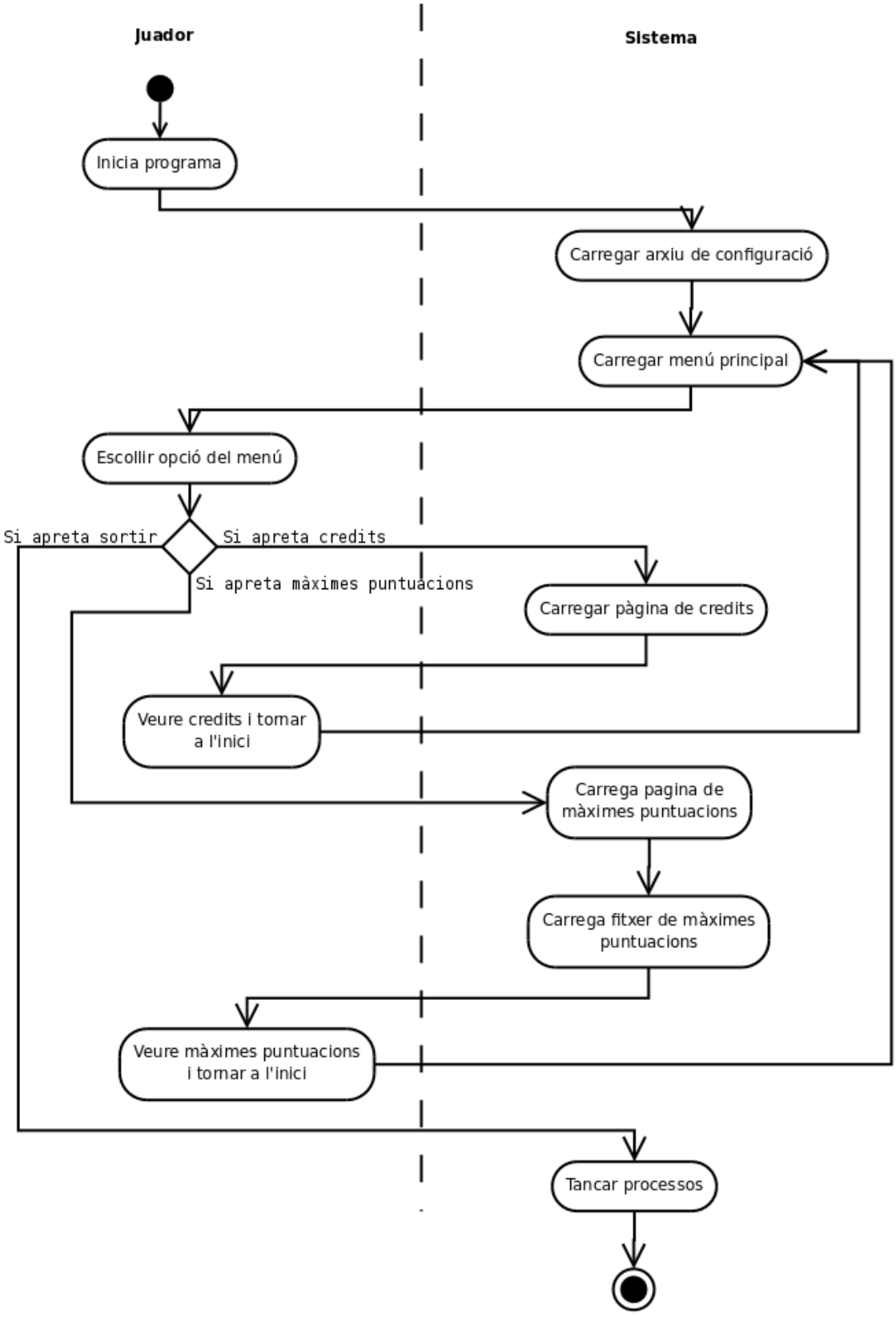


6.2.2. Diagrama d'activitat: Accedir i modificar opcions

Igual que al iniciar partida, primer de tot inicialitzem el programa. Seguidament el sistema s'ocuparà de carregar les opcions de configuració, i mostrar el menú principal per pantalla. Un cop aquí, quan el jugador esculli la opció de configuració, el sistema li mostrarà les opcions, amb les dades per defecte carregades. El jugador podrà modificar aquestes opcions, guardar i/o tornar al menú principal.


Juador
Sistema
Inicia programa
Carregar arxiu de configuració
Carregar menú principal
Escollir opció del menú
Si apreta sortir
Si apreta opcions
Carregar pàgina d'opcions
Agafar opcions de la classe principal carregades a l'inici
Modificar opcions
Si clica torna sense guardar
Si guarda les dades
Guardar opcions al fitxer de configuració
Tancar processos

6.2.3. Diagrama d'activitat: Veure crèdits i màximes puntuacions



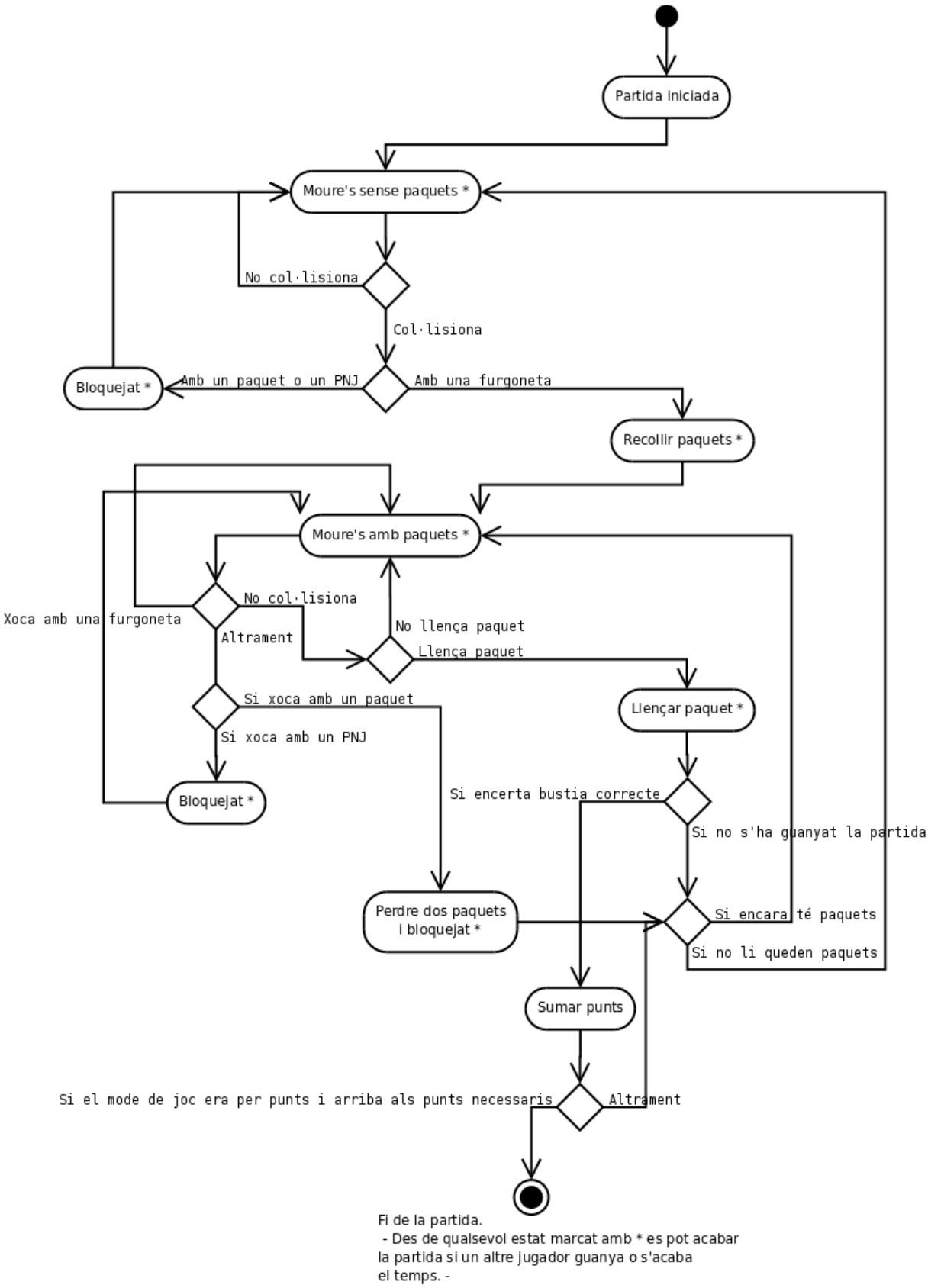
En aquest diagrama veiem les senzilles accions de visualitzar els crèdits així com

visualitzar les màximes puntuacions obtingudes en l'estil de joc individual. Simplement es carreguen les pàgines, a remarcar que les puntuacions màximes s'obtenen de la lectura d'un fitxer, i des de les pàgines tant sols podem tornar al menú principal.

6.2.4. Diagrama d'activitat: Partida

En aquest diagrama s'hi pot observar el diagrama d'activitat referent al desenvolupament de la partida. Remarquem que des de qualsevol dels estats marcats es pot finalitzar la partida a causa que s'hagi acabat el temps o que algun altre jugador hagi arribat al nombre de paquets a entregar.

Aquí podem veure que les activitats per un personatge amb paquets o sense son semblants, però la connexió entre aquestes difereix, per tant, crea dos estats clarament separats. Es comuniquen l'un amb l'altre en l'estat de recollir paquets de la furgoneta, on passem de l'estat sense paquets a l'estat amb paquets, i es tornen a comunicar quan després de llençar un dels seus paquets, o que li caiguin els seus paquets degut a la col·lisió d'un paquet d'un altre jugador, un personatge es queda un altre cop sense paquets.


Partida iniciada
Moure's sense paquets *
No col·lisiona
Col·lisiona
Amb un paquet o un PNJ
Amb una furgoneta
Bloquejat *
Recollir paquets *
Moure's amb paquets *
No col·lisiona
Xoca amb una furgoneta
Altrament
No llença paquet
Llença paquet
Si xoca amb un paquet
Si xoca amb un PNJ
Llençar paquet *
Bloquejat *
Si encerta bustia correcte
Si no s'ha guanyat la partida
Perdre dos paquets i bloquejat *
Si encara té paquets
Si no li queden paquets
Sumar punts
Si el mode de joc era per punts i arriba als punts necessaris
Altrament
Fi de la partida.
- Des de qualsevol estat marcat amb * es pot acabar la partida si un altre jugador guanya o s'acaba el temps. -

7. Disseny del sistema

Després de les fases de descripció de requeriments i anàlisi del sistema, iniciem l'etapa de disseny del sistema. En l'apartat d'anàlisi ens hem centrat en la descripció de les funcionalitats de l'aplicació de forma abstracta. En aquesta etapa de disseny del sistema, s'intenta adaptar la documentació generada en el capítol anterior en vistes a la implementació final del sistema mitjançant diverses eines de programació.

En aquest capítol parlarem sobre el disseny intern d'objectes, atributs i mètodes, i la persistència de l'aplicació.

Cal remarcar que l'aplicació que estem dissenyant no necessita cap mena de manteniment, ja que no existeix cap actor dedicat exclusivament a fer una gestió sobre l'aplicació, així com afegir, modificar i eliminar productes. Per aquest motiu el sistema implementat no és considerat com una aplicació de gestió, per tant no cal que el programari disposi d'una part amb accés restringit per a usuaris de manteniment.

7.1. Diagrames de classes

Els diagrames de classes són els més utilitzats en el modelat de sistemes orientats a objectes. Un diagrama de classes proporciona una visió estàtica del sistema a desenvolupar, ja que mostra les classes que interactuen sense mostrar que passa quan ho fan. En aquest apartat serà detallada l'estructura de classes utilitzada, així com els components de cadascuna de les classes del sistema.

El diagrama de classes amb els atributs i mètodes és molt gran i és impossible visualitzar-lo complert, així doncs es posarà el diagrama de classes sense atributs i mètodes, i tot seguit es mostrarà cada classe per separat on podrem visualitzar-ho amb claredat. A més a més, es comentarà cada classe destacant les principals característiques per tal de fer-les més entenedores.

Ja que a causa del gran nombre de classes utilitzades, un sol diagrama hagués resultat molt gran i al adaptar-lo a un full quedaria il·legible, hem partit aquest en tres figures, una de les quals té la classe inicial amb els diferents estats que representen les possibles pantalles del joc (entenent pantalles com a menús o partides), una altra

conté les classes que intervenen en una partida pròpiament dita, i una tercera figura que conté la part dels controladors de joc.

Després de les figures passarem a explicar el funcionament global anomenant aquelles classes que surten a més d'un diagrama, que són les que ens faran de “pont”.

Nota: Les classes de color beix son classes del motor de videojocs jME (que inclou el motor de físiques enllaçat per nosaltres), i les classes marrons són aportacions d'usuaris a la comunitat per tal de poder integrar fàcilment tots els mòduls.

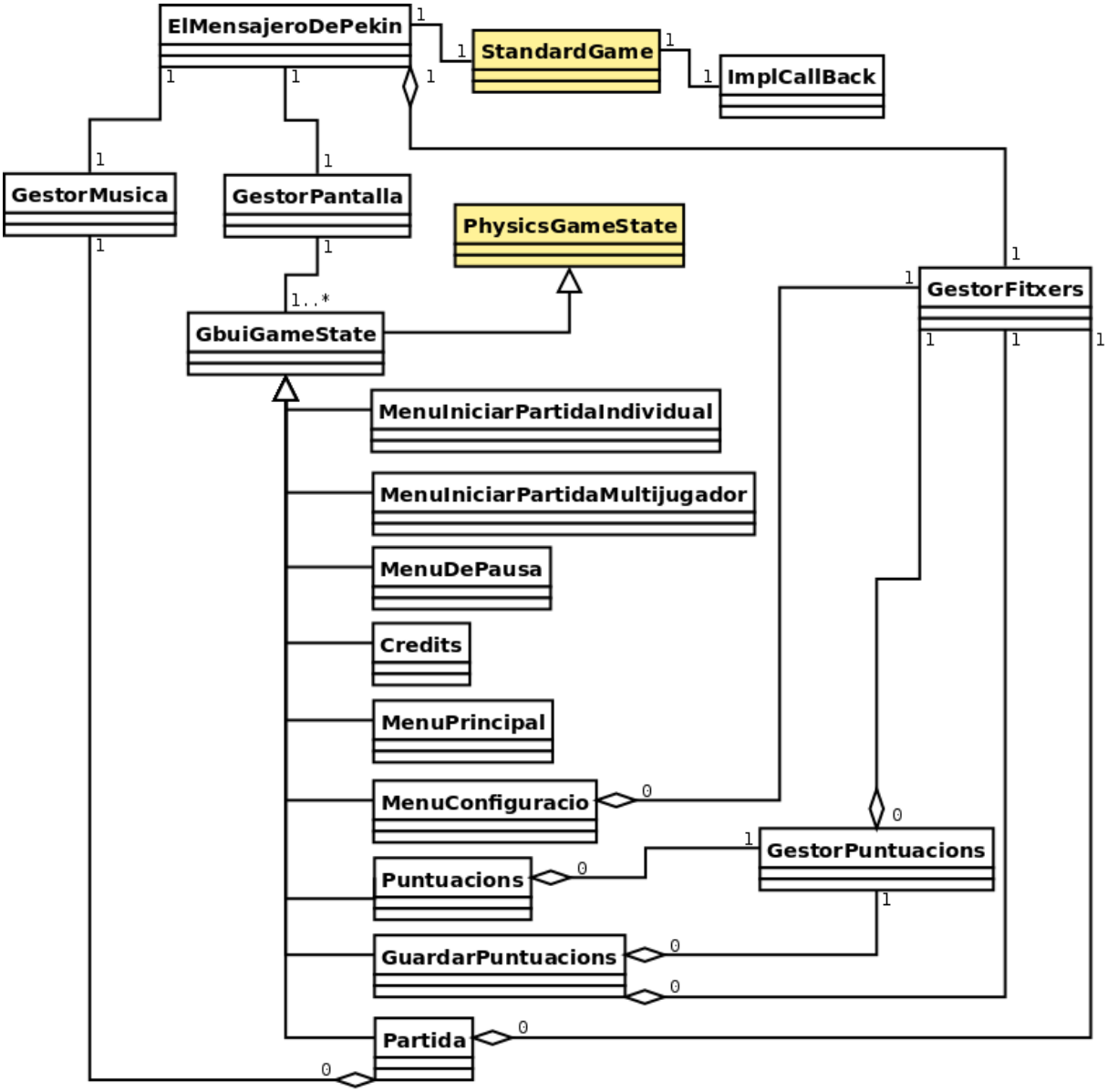


Figura 35: Diagrama de classes parcial en el qual es veuen les pantalles principals

Com veiem en la Figura 35 la classe inicial de l'aplicació, aquella que conté el codi principal, és la classe **ElMensajeroDePekin**. Aquesta classe s'encarrega d'engegar el sistema, i executar un **StandardGame** que és la classe que gestionarà bucle del joc. L'**StandardGame** controlarà la física, en cas de ser necessari, amb la classe **ImplCallBack**.

Per altra banda, al iniciar el joc iniciarem la música mitjançant el **GestorMusica** i carregarem la primera pantalla utilitzant la classe **GestorPantalla**. Aquesta darrera classe és la que iniciarà la pantalla que veurà l'usuari. Creant una llista de **GbuiGameStates**, classe que hereta de la classe del motor de física **PhysicsGameState**, cridarà a la primera pantalla, en aquest cas **MenuPrincipal**. Farem servir el **GestorPantalla** per moure'ns pel menú utilitzant les classes que hereten de **GbuiGameStates** (les explicarem més endavant) fins a començar una partida amb la classe **Partida**.

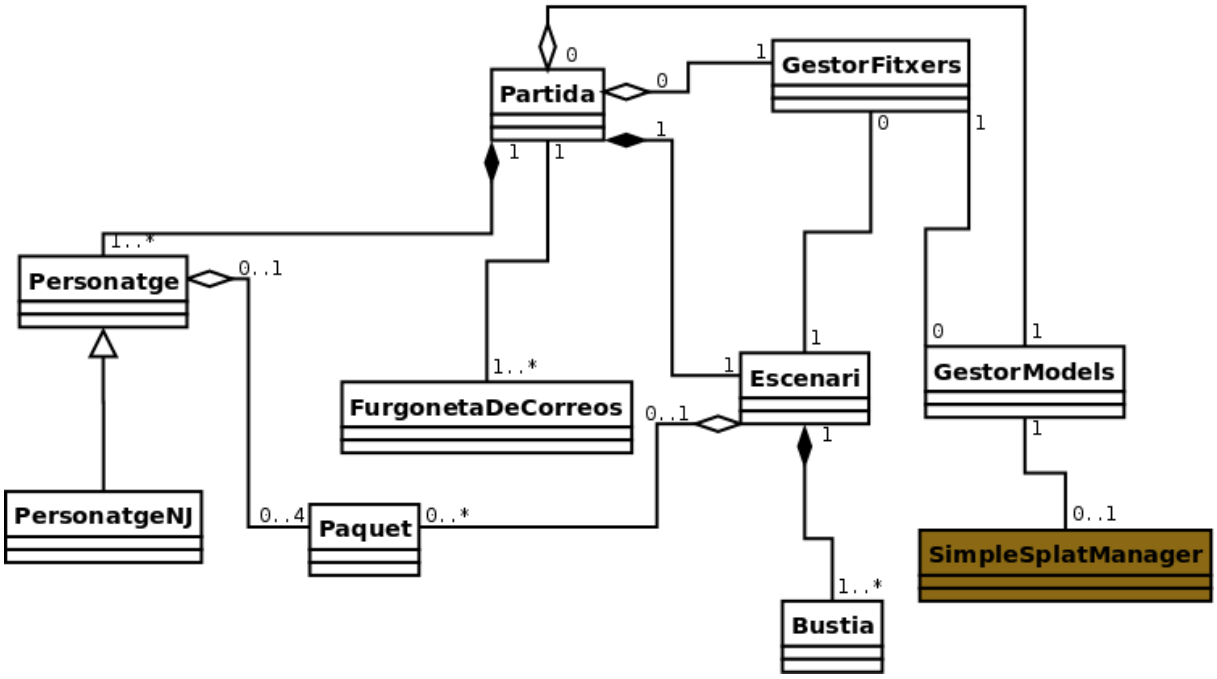


Figura 36: Diagrama de classes parcial en el qual es veuen les classes relacionades amb una partida.

A la Figura 36 veiem el diagrama de classes relacionat pròpiament amb el desenvolupament d'una partida. La classe principal d'aquesta part és **Partida**, que és la mateixa que en la Figura 35 hem descrit com a una herència de

GbuiGameState.

Aquí es veuen tots els elements que composen una partida, com és l'escenari, els personatges, les bústies i demés elements que interaccionaran visualment i físicament amb l'usuari. Totes aquestes classes (**Personatge, FurgonetaDeCorreos, PersonatgeNJ, Paquet, Bustia i Escenari**), són controlades per la classe **Partida**, la qual s'encarrega d'inicialitzar-les i actualitzar-les.

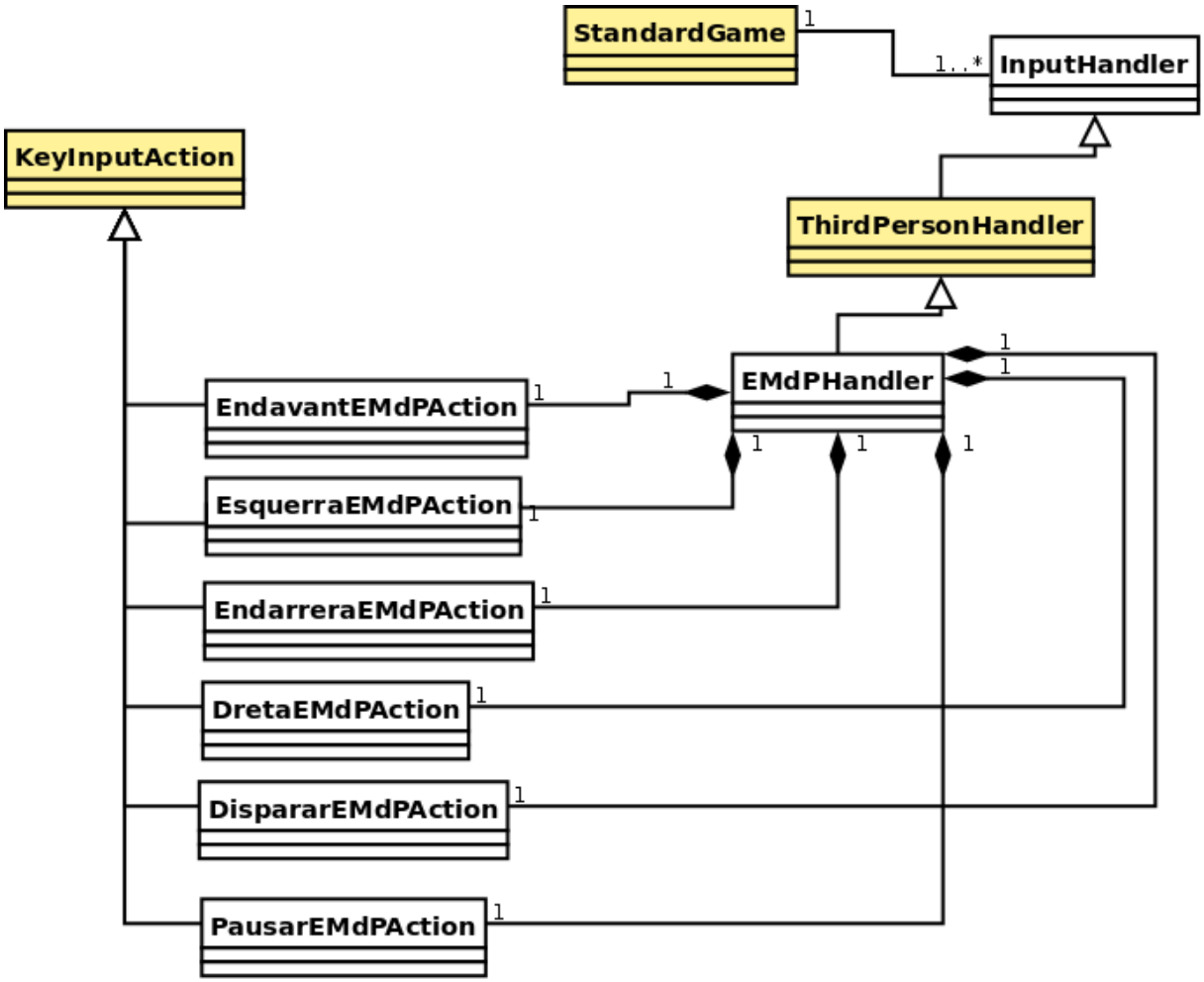


Figura 37: Diagrama de classes parcial en el qual es veuen les classes relacionades amb els controladors.

Finalment, en la Figura 37 veiem la darrera part del diagrama de classes general, pertanyent a la part dels controladors del joc.

Es pot veure que la classe **StandardGame** és la que controla l'entrada del **InputHandler**, ambdues classes pertanyents al java Monkey Engine. Mitjançant la classe **EmdPHandler** que hereta de **ThirdPersonHandler** (també pertany al jME), indiquem a l'**StandarGame** quines accions ha de fer servir quan es premi una tecla

prèviament definida.

Cada una de les classes que extenen de **KeyInputAction** representa una acció que pot fer el jugador.

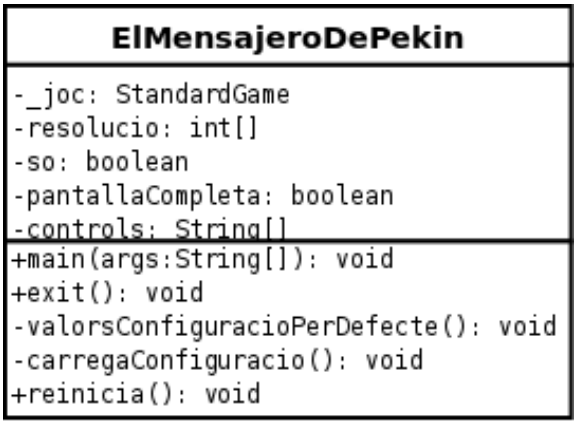
7.1.1. Descripció de les classes

A continuació descriurem les classes representades en les Figures 35, 36 i 37 de manera més extensa i mostrant el diagrama de cada classe, posant en ell els mètodes i atributs més importants que li corresponen, ja que hi ha classes que tenen un gran quantitat d'atributs i mètodes necessaris.

7.1.1.1. Classes parcials del diagrama de classes relacionat amb les pantalles principals

Aquí explicarem les classes que es poden veure en el diagrama de la Figura 35.

7.1.1.1.1. ElMensajeroDePekin



Aquesta és la classe “main” del nostre programa, aquella que inicia el joc i s'ocupa de que s'hi posin les dades bàsiques per a que comenci a funcionar. També s'ocupara, quan arribi el cas, de tancar l'aplicació.

A més, guarda la informació bàsica de configuració del nostre sistema, carregada des del fitxer de configuració.

Components de la classe:

- Atributs:

- **_joc**: variable que conté l'**StandardGame** principal que executa el joc.
- **resolucio**: vector de dos valors amb la resolució de la pantalla.
- **so**: variable booleana que diu si el so està activat o no.
- **pantallaCompleta**: variable booleana que diu si treballarà amb pantalla completa.
- **controls**: vector de cadenes de caràcters que guarda el nom dels controladors.

- Mètodes:
  - **main**: mètode principal que inicia el programa i el joc.
  - **exit**: mètode que tanca el joc.
  - **valorsConfiguracioPerDefecte**: mètode que posa els valors de configuració per defecte que estan guardats al fitxer de configuració.
  - **carregaConfiguracio**: carrega la configuracio que tenen les variables de classe a l'aplicació.
  - **reinicia**: reinicia el joc i carrega els nous valors de configuració.

7.1.1.1.2. StandardGame

Aquesta classe pertany al jMonkey Engine. És la classe que gestiona el funcionament del joc: Inicialització, visualització, el *loop* d'actualització, etc.

La fem servir per a crear el joc, i hi afegim els estats amb les que treballem posteriorment. Dintre dels estats, fem servir els metodes d'*update* i *render* perquè el **StandardGame** ho executi quan calgui.

Aquesta classe es pot veure detallada a la Figura 15 de la pàgina 40.

7.1.1.1.3. ImplCallBack

La classe **ImplCallBack** és la classe que inicialitza el sistema Physics, afegint una funció de callback, per quan succeeixin unes determinades col·lisions dinàmiques. Les col·lisions que intervenen nodes dinàmics estan dividides en dos grans blocs, les que hi intervé un personatge, o les que hi intervé un paquet:

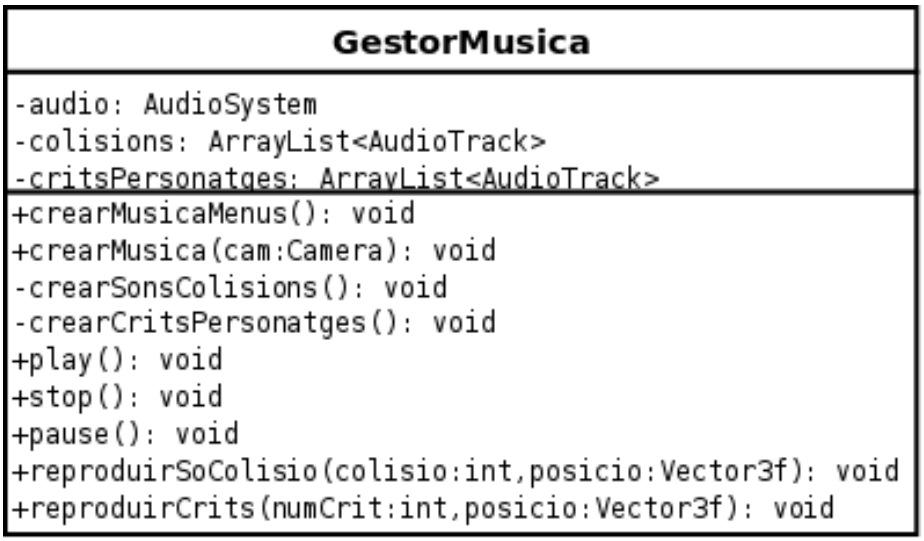
- Col·lisions que hi intervé un paquet:
  1. **Paquet contra Personatge**, El personatge perd 2 paquets (si els té), i passa a estar bloquejat durant 2 segons.
  2. **Paquet contra PNJ**, El Personatge no jugador, passa a estar bloquejat durant 2 segons, fet que deixa de perseguir a personatges.
  3. **Paquet contra Bústia Bona**, bústia bona vol dir que la bústia que xoca amb el paquet és del mateix color. En aquest cas, el jugador sumaria un punt, i el paquet s'autodestruiria.
  4. **Paquet contra Bústia Dolenta**, bústia dolenta, significa que no és del mateix color la bústia i el paquet. Per tant, en aquest cas, la bústia llançarà el paquet rebut en direcció cap al personatge que l'ha llançat. Si xoquen, passem a col·lisió 1, si no passem a col·lisió 5.
  5. **Paquet contra Escenari**, com a escenari s'entén qualsevol element visual no descrit anteriorment i que pugui considerar-se part de la zona de joc. En aquest altre cas, el paquet s'autodestrueix.
- Col·lisions que hi intervé un personatge:
  1. **Personatge contra PNJ**, el Personatge queda bloquejat segons un atribut del **PNJ**.
  2. **Personatge contra FurgonetaDeCorreos,** si el personatge no té paquets i la furgoneta encara té algun carregament, el personatge passa a estar bloquejat quatre segons per tal de descarregar la furgoneta, i el seu estat serà immune, per tal de que els altres jugadors no li puguin llançar

paquets.

Components de la classe:

- Mètodes:
  - **ImplCallBack:** Constructor que inicialitza el motor de físiques. Li passem per paràmetre el rootNode que és el que conté tots els elements visuals i l'espai físic.

7.1.1.1.4. GestorMusica



La classe **GestorMusica**, és l'encarregada de controlar la música durant tota l'aplicació. Un videojoc no només té un reproductor de música, que simplement reprodueix cançons, si no que també té efectes sonors. Aquests seran sons reproduïts des de un punt concret de l'escenari aconseguint així un efecte de so en tres dimensions.

Aquesta classe s'ocuparà de gestionar el tema de l'àudio. Els sons que ha de reproduir són:

- Música durant el joc.
- Música durant els menús.
- Efectes sonors de col·lisions
- Crits que fan els jugadors al llançar un paquet.

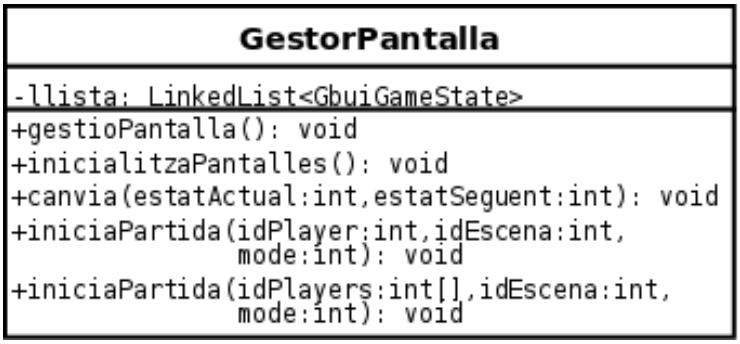
A part, com és pot veure, també tindrà mètodes per tal de reproduir, pausar o parar la música, tant del joc com dels menús.

Components de la classe:

- Atributs:
  - **audio:** Conté l'àudio del sistema, per tal de poder-lo configurar com l'usuari desitgi.
  - **colisions:** És una cadena de pistes d'àudio, que contenen els sons que s'han de reproduir segons quin tipus de col·lisió hagi succeït.
  - **critsPersonatges:** És una cadena que conté els crits dels personatges, ja que cada personatge tindrà el seu propi crit.
- Mètodes:
  - **crearMusicaMenus:** Mètode que simplement inicialitza la música i la reprodueix, creant una llista de reproducció, i una opció que repeteixi sempre que finalitzi.
  - **crearMusica:** Creem la música que sonarà durant la partida, així com inicialitzem els sons de col·lisions i els sons dels crits.
  - **crearSonsColisions:** Mètode privat que crearà els sons de les col·lisions.
  - **crearCritsPersonatges:** Mètode privat que crearà la llista de crits que faran els personatges a l'hora de llançar un paquet.
  - **Play:** Si la música de fons està parada o pausada, torna a reproduir.
  - **Pause:** Si la música s'està reproduint, canvia a mode pausat.
  - **Stop:** Si la música s'està reproduint, para-la.
  - **reproduirSoColisio:** Reproduïm un so del llistat de col·lisions, depenent de l'id de col·lisió que ens passin. A part el so és tridimensional, per tant també em de col·locar-lo a un punt 3D.
  - **reproduirCrits:** Reproduïm un so del llistat de crits, depenent de l'id de personatge que ens passin. A part el so és tridimensional, per tant també

em de col·locar-lo a un punt 3D segons la posició del jugador.

7.1.1.1.5. GestorPantalla



La classe **GestorPantalla** és la que servirà per a gestionar les “pantalles” que s'han d'iniciar o finalitzar.

Bàsicament, el que fa és crear una llista amb tots els **GbuiGameState** que farem servir, i canviar l'estat d'actiu entre ells quan és cridada. A més, en casos especials com iniciar partida, també s'encarrega de crear-ne l'objecte i gestionar els paràmetres.

Components de la classe:

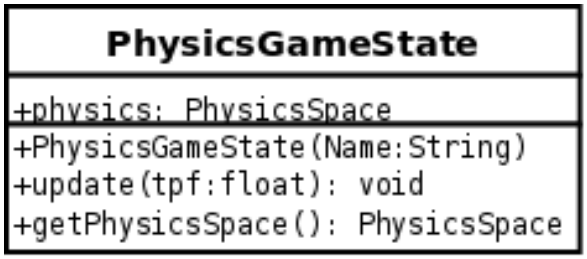
- Atributs:
  - **llista**: llista de **GbuiGameState** per a guardar tots els nostres estats
- Mètodes
  - **gestioPantalla**: mètode que crida a la creació de la llista de **GbuiGameStates** i afegeix els estats al **GameStateManager**.
  - **inicialitzaPantalles**: crea la llista de **GbuiGameStaes**.
  - **canvia**: canvia entre dos estats que li passem per paràmetre, desactivant-ne un i activant l'altre.
  - **iniciaPartida**: mètode que crea i inicia una partida.

7.1.1.1.6. GestorFitxers

La classe **GestorFitxers** és la encarregada de treballar amb els fitxers de l'aplicació

que guarden dades (els fitxers **.emdp**). Les tasques bàsiques són llegir i escriure els fitxers de la manera que calgui en cada situació, sigui la configuració global, llegir com és un **Personatge**, etc.

7.1.1.1.7. PhysicsGameState

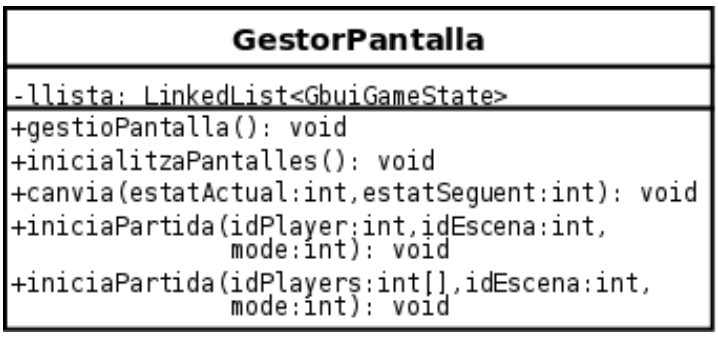


La classe **PhysicsGameState** que hereta de **BasicGameState**, és la classe encarregada de controlar la física en els estats del joc. Serà l'encarregada de controlar els nodes dinàmics de l'estat actual, controlant la posició i rotació quan xoquin amb un altre node dinàmic, o amb un node estàtic.

Components de la classe:

- Atributs:
  - **physics:** Conté l'espai físic, per tal de poder crear nodes estàtics, dinàmics, ...
- Mètodes:
  - **PhysicsGameState:** Constructor, que assignem un nom. Crida al constructor pare.
  - **getPhysicsSpace:** Getter que ens retorna l'atribut physics.
  - **update:** Mètode que actualitza el **GameState**, passant també per actualitzar les físiques d'aquell instant en que ha estat cridat.

7.1.1.1.8. GbuiGameState



Aquesta és la classe amb la qual gestionem els estats del videojoc. Ja que el nostre joc funciona amb estats, i tots els nostres estats compartien una serie de variables i funcionalitats, vam crear aquesta classe per tal d'englobar-les a totes.

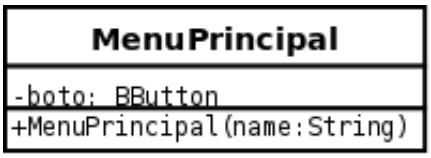
Aquesta classe hereta, per tant, d'un **GameState**. En aquest cas de **PhysicsGameState** ja que necessitàvem que així fos per a poder executar la partida.

Per tant, aquesta classe és l'encarregada de crear els estats i durant el transcurs de l'aplicació gestionar el canvi d'aquests.

Components de la classe:

- Atributs:
  - **display**: es refereix a una instància del **DisplaySistem** del joc, que guardem en una variable per treballar-hi més còmodament.
  - **window**: objecte que guarda i mostra tot allò relacionat amb la interfície gràfica del sistema GBUI.
- Mètodes:
  - **GbuiGameState**: constructor que inicialitza el sistema del GBUI i crea la finestra.
  - **activate**: mètode que activa un estat inactiu, afegint a l'arbre de nodes el node principal d'aquesta classe.
  - **deactivate**: mètode que s'encarrega de parar un estat i deshabilitar les funcions del la interfície gràfica per a que no tinguin afecte fins que es torni a activar.

7.1.1.1.9. MenuPrincipal



Encarregada de mostrar el menú principal, aquesta és la classe d'estat que primer carrega el sistema. **MenuPrincipal** hereta de **GbuiGameState** i utilitzant l'atribut **window** comentat anteriorment, fa la funció de mostrar els botons bàsics per a començar el joc.

Components de la classe:

- Atributs:
  - **boto**: essent aquest una instància de la classe botó del nostre GUI, farem servir un sol objecte per a crear tots els botons, ja que després de afegir-lo a la finestra l'objecte en sí ja no servirà de res, ja que per a modificar-lo haurem d'accedir directament al fill de l'objecte **window.** Per tant, per cada botó que necessitem crearem un objecte **boto** i l'afegirem a la finestra.
- Mètodes:
  - **MenuPrincipal**: és el constructor de la classe. L'únic que farem serà crear sempre els mateixos botons quan engeguem l'aplicació de forma estàtica, en el sentit que la forma d'aquests botons no canviarà mai. Per tant, aquest mètode és el més important perquè és el que posa cada botó al seu lloc.

7.1.1.1.10. MenuIniciarPartidaIndividual

Aquesta classe és la que mostra les opcions a escollir per a jugar una partida d'un sol jugador. Un cop escollides les opcions i acceptades, serà la que passi les dades necessàries als gestors per a poder crear la partida.

Igual que la resta de menús, és una herència de **GbuiGameState**, i és una classe bastant rígida ja que el menú a mostrar sempre serà el mateix, encara que en aquesta ja apareixen coses més dinàmiques com la selecció de pantalla o jugador.

Components de la classe:

- Atributs:
  - **boto**: objecte que s'encarrega de representar els botons genèrics del menú. Un cop instanciat, l'afegim a la finestra i el fem servir per a crear més botons.
  - **text**: objecte **BLabel** que fem servir per a escriure tot el text que necessitem renderitzar.
  - **contenidor**: objecte genèric del nostre GUI que serveix per a crear-hi imatges com a *background* i poder-les fer interactuar amb altres elements de la mateixa GUI.
  - **toggle**: botó de dos estats, activat i desactivat, per a gestionar el mode de joc que farem servir.
  - **idPantalla**: identificador de la pantalla. Inicialment a zero, quan comencem el joc llegirà el valor que li hem passat per paràmetre al creador per a cridar el constructor de la partida.
  - **IdPlayer**: identificador del jugador. Al igual que idPantalla, es crea a zero i quan comença la partida s'agafa el valor passat per paràmetre per a saber amb quin personatge es jugarà.
  - **Mode**: variable que simbolitza el tipus de joc que farem, per paquets o bé per temps.
- Mètodes:
  - **MenuIniciarPartidaUnJugador**: constructor de la classe. És el mètode principal que crea la pantalla per a seleccionar les opcions.
  - **creaMenuPantalles**: aquest mètode s'encarrega de la part de crear el seleccionador de pantalles, tant la imatge que les representa com les fletxes per a navegar-hi, fent servir el mètode **posaBotonsAbansDespres**.
  - **crearMenuPersonatge**: aquest mètode té el mateix funcionament que l'anterior però per als personatges.

- **crearMenuModalitat**: aquest mètode s'encarrega de crear els botons d'estats que seleccionaran el mode de joc.
- **posaBotonsAbansDespres**: aquest mètode s'encarrega de crear les fletxes de navegació que permetran seleccionar els personatges i les pantalles.

7.1.1.1.11. MenuIniciarPartidaMultijugador

Aquesta classe és similar a l'anterior, però serveix per a iniciar partides de varis jugadors, podent ésser aquestes des de dos fins a quatre. Igual que l'anterior, permet definir una sèrie de paràmetres que seran passats al crear la partida, i hereta de **GbuiGameState**, ja que com tots els menús, és un estat més.

7.1.1.1.12. MenuConfiguracio

Aquesta classe s'encarrega de carregar el menú de configuració i cridar al gestor de configuració quan hi ha modificacions que es volen gravar, passant les dades necessàries que recull de les opcions.

Igual que totes les classes de menú, hereta de **GbuiGameState,** de on agafa les variables bàsiques per a mostrar per pantalla.

Components de la classe:

- Atributs:
  - **checkBox**: objecte de la classe **BCheckBox** de la implementació del nostre GUI que serveix per a crear botons d'activat/desactivat. Igual que en la classe **MenuPrincipal**, tots els objectes d'aquesta classe son reutilitzats tants cops com faci falta, ja que al afegir-los a la finestra ja no necessitem més la instància de l'objecte.
  - **text**: objecte de la classe **BLabel** que fem servir per a escriure text per pantalla en diferents punts de la classe.
  - **boto**: objecte de la classe **BButton** per a afegir els botons de confirmar i cancel·lar al menú.
  - **llista**: objecte de la classe **BList** que fem servir quan donem a escollir la

resolució.

- **comboBox**: objecte de la classe **BComboBox** que serveix per a crear llistes desplegables. Es fa servir, en aquest cas, per a crear les quatre llistes que permetran escollir els controls.

- Mètodes:
  - **MenuConfiguracio**: és el constructor de la classe, que s'encarrega de omplir l'objecte **window** i de gestionar que s'ha de fer amb les dades quan el botó de guardar es premut.
  - **creaLlistaControls**: aquest mètode és cridat des del **MenuConfiguracio** i s'encarrega de crear, només, les llistes de controladors per a cada jugador.
  - **crearLlistaResolucio**: s'encarrega de crear els botons seleccionables per a canviar la resolució.
  - **crearCheckBoxAudio**: crea el selector per a activar i desactivar el so.
  - **crearCheckBoxPantallaCompleta**: aquest mètode és l'encarregat de crear el selector per a activar o desactivar la pantalla completa.

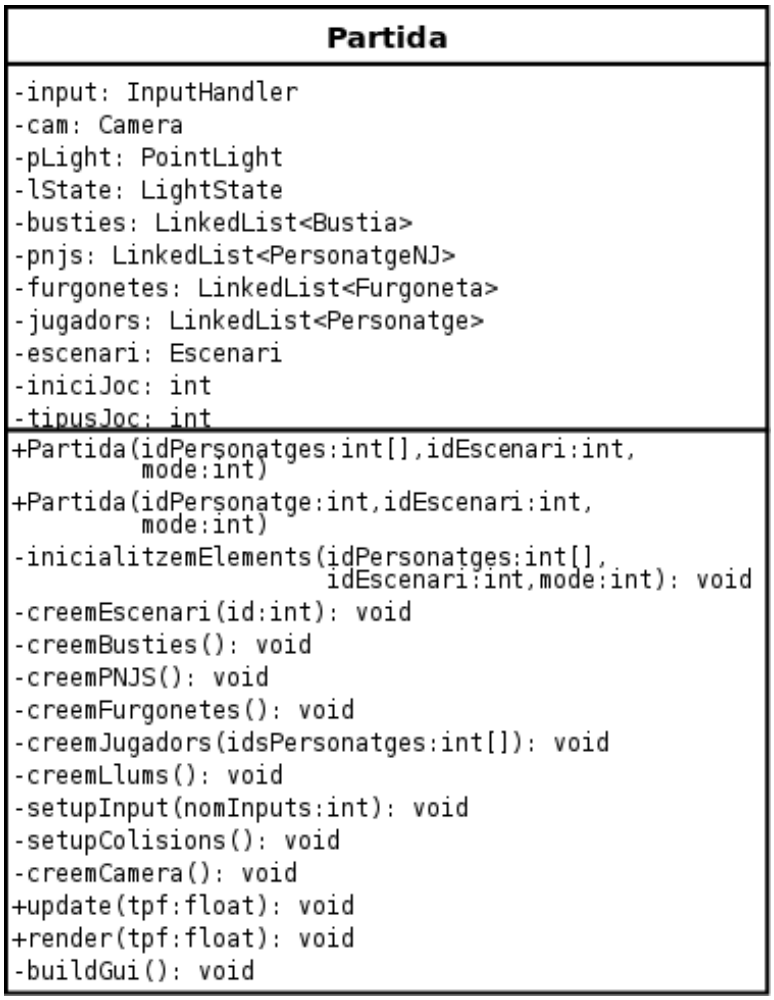
7.1.1.1.13. Credits

Aquesta és una classe molt bàsica que serveix per a mostrar una llista de crèdits estàtica per pantalla.

7.1.1.1.14. Puntuacions

Aquesta classe recull les màximes puntuacions que es troben en els fitxers de registres, gràcies a la classe **GestorFitxers**, i en crea una llista per a ser mostrada per pantalla amb les màximes puntuacions.

7.1.1.1.15. Partida



La classe **Partida** és la encarregada de crear i gestionar una partida del joc. Fent una instància d'aquesta classe creem la partida, passant-li el número de personatges, l'escenari i el mode de joc. Dintre la classe es crearan els personatges, bústies, escenari i demés com a objectes, agafant les dades dels fitxers de configuració necessaris, i es començarà la partida.

Components de la classe:

- Atributs
  - **input**: vector de llargada quatre elements que conté les entrades dels quatre jugadors (o d'aquells jugadors que estiguin jugant).
  - **cam**: càmera del joc. És fixe, i l'iniciem al començar la partida.

- **lState**: estat que controla la llum de la pantalla. S'hi afegeixen objectes **PointLight** que definiran els punts de llum.
- **pLight**: **PointLight** de llum ambient que fem servir per a il·luminar l'escenari.
- **busties**: llista de bústies de l'escenari.
- **furgonetes**: llista de furgonetes de l'escenari.
- **pnjs**: llista de personatges no jugadors per a l'escenari.
- **jugadors**: llista de personatges jugadors que jugarà aquesta partida.
- **escenari**: objecte de tipus **Escenari** que representa l'escenari a on juguem.
- **iniciJoc**: guarda el valor del *timer* quan comencem la partida.
- **tipusJoc**: conté el mode de joc de la partida, per temps o per paquets (definits a la classe GestorConstants).

• Mètodes

- **Partida**: constructor de la classe, ajusta les variables passades per paràmetre per a que siguin en el format del métode *inicialitzemElements* i, finalment, el crida.
- **inicialitzemElements**: crida els mètodes necessaris per a iniciar tots els elements del joc, els guarda en les variables, captura l'instant final de timer i es comença el joc.
- **creemEscenari**: crea l'escenari i l'afegeix al *rootNode*.
- **creemBusties**: crea les bústies (tantes com tingui definit l'escenari que en tindrà) i les afegeix a l'escenari i a la llista de bústies.
- **creemPNJS**: crea els personatges no jugadors, els afegeix a l'escenari i a la variable llista de la classe.
- **creemFurgonetes**: seguint les indicacions de configuració de l'escenari, crea les furgonetes i les afegeix a la llista i a l'escenari.

- **creemJugadors**: crea tants jugadors com li hàgim dit al constructor, i els afegeix a la llista i a l'escenari, en la posició que porta per configuració.
- **creemLlums**: crea la llum, l'afegeix al **LightStae** i actualitza el *renderState* del *rootNode*.
- **setupInput**: inicialitza els mètodes d'entrada per a cada usuari i els afegeix a l'*input*.
- **setupColisions**: inicialitza el sistema de col·lisions.
- **creemCamera**: inicialitza la càmera i la col·loca al seu lloc (d'on no es mourà, ja que és fixe).
- **update**: mètode que actualitza les dades del joc en cada iteració d'aquest (*game loop*).
- **buildGui**: mètode que crea els textos de pantalla que porten el comptador de paquets, punts, etc.

7.1.1.1.16. MenuDePausa

La classe **MenuDePausa** és la que s'executa al pausar una partida durant la seva evolució prement la tecla F10.

Bàsicament, ha de tenir la funcionalitat de sortir de la partida i tornar al menú principal, o bé retornar-hi en el mateix estat que es va deixar.

7.1.1.1.17. GuardarPuntuacions

Aquesta classe pertany als **GameStates**, i és la que, al finalitzar una partida, mostra qui és el vencedor, la puntuació, i si estarà o no en el registre de màximes puntuacions.

A més, a través de la classe **GestorFitxers** guardarà les dades, si cal, per a actualitzar les les màximes puntuacions.

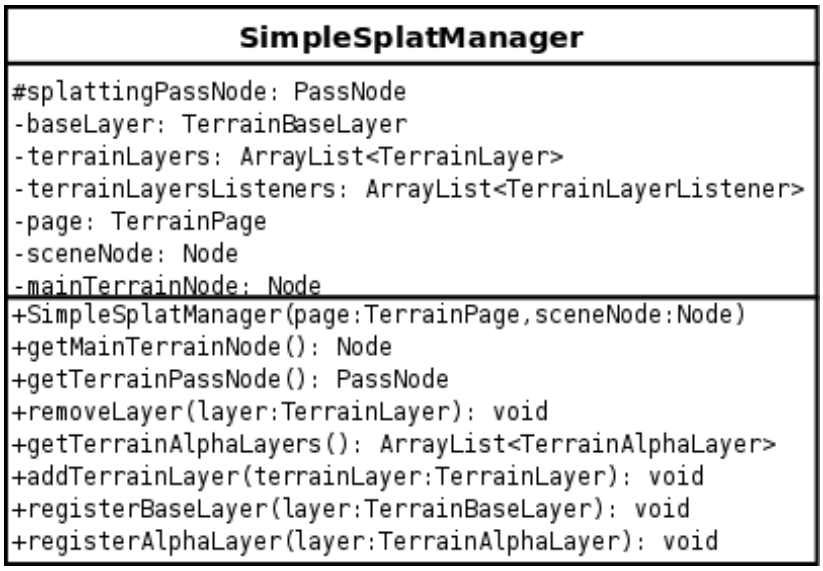
7.1.1.2. Classes parcials del diagrama de classes relacionat amb una partida

Aquí s'explicaran les classes relacionades amb la Figura 36.

7.1.1.2.1. GestorModels

La classe **GestorModels** és la que s'encarrega de llegir els models i posar-los en nodes per a que hi puguem treballar. A més, també té funcionalitats com carregar textures o carregar els escenaris, que és una combinació de les dos funcionalitats anteriorment esmentades.

7.1.1.2.2. SimpleSplatManager



Aquesta classe ha estat subministrada per un usuari de la comunitat del jME i del MW3D (Monkey World 3D) per tal de poder importar terrenys generats amb el MW3D al jME sense tenir problemes, ja que al començament l'importador tenia errors. Nosaltres, l'únic mètode utilitzat ha sigut el constructor i posteriorment el mètode **registerBaseLayer.**

7.1.1.2.3. Escenari

La classe Escenari, igual que la classe **Bústia**, **Paquet**, etc., per tal de poder fer l'estructura d'arbre de nodes, hereterà de la classe Node. A part, aquesta classe serà la classe pare de tots els elements visuals, així com els jugadors, personatges no jugadors, el model de l'escenari,etc. A més a més per tal de poder controlar la posició d'aquests objectes, la classe Escenari contindrà atributs per tal de definir les posicions. També per tal de poder controlar les físiques contindrà un node dinàmic.

Components de la classe:

- Atributs:
  - **id**: Número que ens indica quin escenari és.
  - **Nom**: Nom que té aquest escenari per tal de mostrar-lo als jugadors.
  - **Descripció**: Petita descripció de l'escenari que també es mostrarà als jugadors.
  - **Model**: Model 3d de l'escenari.
  - **FotoInicial**: Conté la ruta de l'imatge que és mostra per tal de seleccionar un escenari.
  - **PosBusties**: Llistat de les posicions de totes les bústies
  - **PosPNJS**: Llistat de les posicions dels Personatges no jugadors.
  - **PosFurgonetes**: Llistat de la posició inicial de totes les furgonetes
  - **PosJugadors**: Llistat de la posició inicial de tots els jugadors.
- Mètodes:
  - **Escenari**: constructor de l'escenari, carrega el model d'aquest així com totes les posicions dels objectes que conté.
  - **GetNumBusties**: retorna el número de busties que conté la llista posBusties.
  - **GetNumPNJS**: retorna el número de personatges no jugadors que conté la llista posPNJS.
  - **GetNumFurgonetes**: retorna el número de furgonetes que conté la llista posFurgonetes.
  - **GetNumJugadors**: retorna el número de jugadors que conté la llista posJugadors.
  - **CrearEscenariEstàndar**: mètode per quan no li passem un model a l'escenari, que carregarà un escenari estàndard definit per codi.

- **CrearFisiques**: mètode que crea la física de l'escenari.

7.1.1.2.4. Bustia

Aquesta classe representa l'objecte bústia. Aquesta serà l'encarregada de rebre l'impacte dels paquets. Per saber si el paquet és bo o dolent, haurà de tenir un atribut que defineixi de quin color és. Igual que la classe **Paquet,** per tal de poder definir un pare de la bústia, aquesta classe heretarà de la classe **Node.**

Components de la classe:

- Atributs:
  - **nom:** contindrà el nom de la bústia.
  - **Model:** guardarà el model gràfic de la bústia.
  - **Color:** atribut que defineix de quin color és.
  - **NodeEstàtic:** per tal de controlar la física, la bústia tindrà un node estàtic.
- Mètodes:
  - **Bústia**: constructor que inicialitza tots els atributs de la bústia, en alguns casos, és l'encarregat de cridar al mètode per carregar models 3d de la classe **GestorModels**.

7.1.1.2.5. FurgonetaDeCorreos

Aquesta classe serà l'encarregada de controlar el repartiment de paquets entre els jugadors, així com d'actualitzar la posició d'aquestes dins de l'escenari. A més també tindrà atributs que defineixin la part gràfica, guardant el model 3D. Igual que les classes anteriors **Paquet**, **Bústia**, ... Per tal de poder definir el pare de la **FurgonetaDeCorreos**, aquesta classe heretarà de la classe **Node.**

Components de la classe:

- Atributs:
  - **model:** Contindrà el model 3d que defineix la furgoneta.
  - **Carregaments:** Indicarà el número de carregaments que portarà aquesta

furgoneta.

- **PuntArribada:** Tenim aquest atribut per tal de indicar a quin punt és farà el descarregament dels paquets.
- **PuntFinal:** Un cop la furgoneta ha descarregat els paquets, anem a aquest punt, a on ens esperarem fins a nova ordre.
- **NodeEstàtic:** Node per controlar les físiques.
- **TempsBlocat:** Atribut per tal de poder bloquejar una furgoneta per un cert temps.

- Mètodes:
  - **FurgonetaDeCorreos:** Constructor que inicialitza els atributs de la classe, i carrega el model 3d.
  - **Actualitzar:** Mètode que controla el posicionament de la furgoneta a dins de l'escenari, tinguen en compte si està al punt de descàrrega, al punt inici, si té paquets,...
  - **anarA:** A partir d'aquest mètode fem moure la furgoneta cap a una direcció. Aquest mètode el cridarà el mètode *actualitzar*, a on, segons si tenim paquets i segons la posició actual, li indicarà cap a on ha de dirigir-se la furgoneta.
  - **PotDescarregar:** Mètode que indicarà si podem fer una descàrrega, ja sigui perquè estem a la posició o per controlar si tenim paquets.
  - **UnCarregamentFet:** Decrementem el número de paquets, i bloquegem la furgoneta x segons, que representarà que el personatge està descarregant els paquets.

7.1.1.2.6. Personatge

Aquesta classe representa als personatges. Conté tant la part gràfica com física d'aquests, contenent llavors tant atributs físics, com l'estructura de nodes i malles que el componen, i l'estructura de *bones* que fem servir per a animar al personatge.

7.1.1.2.7. PersonatgeNJ

Aquesta classe hereta de la classe **Personatge.** Aquesta última com hem explicat anteriorment és la que conté tant la part gràfica, com física dels personatges. És a dir, conté atributs que conten el model, textura, etc. i també té atributs que contenen la força i velocitat. La classe **PersonatgeNJ** a part d'heretar tots aquests atributs, també s'haurà d'encarregar de controlar la posició del **PersoantgeNJ** a dins de la pantalla, perseguint els jugadors si els detecta.

Components de la classe:

- Atributs:
  - **visio:** La distància màxima en que els personatges no jugadors poden veure a algun jugador.
- Mètodes:
  - **PersonatgeNJ:** Constructor que inicialitza el pare del PersonatgeNJ (classe Personatge) i assigna la visió que tindrà aquest.
  - **Actualitzar:** Mètode que actualitza la posició en el mapa segons si ha detectat un jugador.
  - **HiHaJugadorDavant:** Mètode que llança un raig endavant i retorna cert o fals si hi ha un jugador a davant.
  - **BuscarAlVoltant:** Busca en totes les direccions si hi ha un jugador.
  - **EsFillDUnPersonatge:** Ens indica si el node trobat és fill d'un Personatge.

7.1.1.2.8. Paquet

La classe **Paquet**, és la que representa l'objecte paquet. Com hem explicat anteriorment, el personatge podrà llançar paquets per l'escenari, podent xocar i rebotar contra els objectes que contingui la pantalla. Per poder fer tot això la classe **Paquet** contindrà un DynamicPhysicsNode, que serà l'encarregat de controlar tota la física del node. A part de la física, la classe **Paquet** també haurà de indicar de quin color és i de quin jugador és.

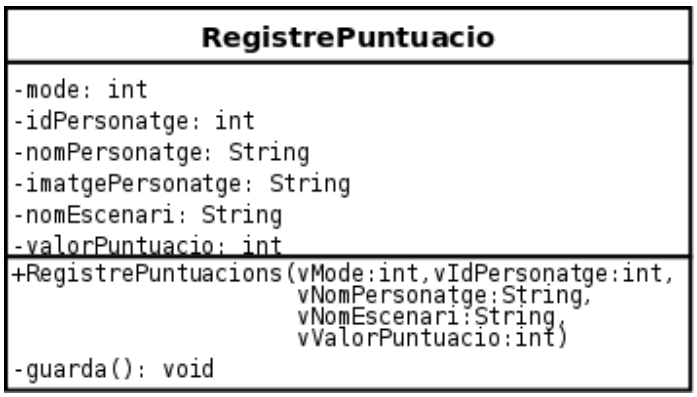
Per tal de poder fer que un paquet sigui fill, en el graf d'escena, d'un jugador, la

classe **Paquet** serà una herència de la classe **Node.**

Components de la classe:

- Atributs:
  - **nom:** contindrà el nom del paquet.
  - **Model:** el model que tindrà el paquet gràficament.
  - **Color:** indicarà de quin color és el paquet.
  - **NodeFísic:** serà el node encarregat de controlar la física dels xocs i els rebots del paquet.
  - **IdJugador:** indicarà a quin jugador pertany.
- Mètodes:
  - **Paquet:** Constructor que inicialitzarà els atributs del paquet.
  - **SetMaterial:** Mètode per canviar el material del paquet. El material defineix els atributs físics del paquet, com el pes, els rebots, la densitat,...

7.1.1.2.9. RegistrePuntuacio



Aquesta classe representa un registre de puntuació de final d'una partida. Es fa servir per a interactuar amb la classe **GestioFitxers** i l'estat **GuardarPuntuacions** per a crear i guardar els registres que s'obtenen al final de la partida

Components de la classe:

- Atributs:
  - **mode**: tipus de joc al que pertany la puntuació.
  - **idPersonatge**: personatge al que pertany la puntuació.
  - **nomPersonatge**: nom del personatge.
  - **imatgePersonatge**: imatge del personatge (per a mostrar a la pàgina de veure puntuacions).
  - **nomEscenari**: nom de l'escenari a on s'ha aconseguit la puntuació.
  - **valorPuntuacio**: valor de la puntuació aconseguida (sigui per temps o per paquets, sempre és un enter)
- Mètodes:
  - **RegistrePuntuacio**: creador de la classe. Li assigna els valors bàsics passats per paràmetre.
  - **guarda**: mètode que guarda un registre al fitxer corresponent.

7.1.1.3. Classes parcials del diagrama de classes relacionat amb els controladors

Aquí s'explicaran les classes relacionades amb el diagrama de classes de la Figura 37.

7.1.1.3.1. InputHandler

Aquesta classe forma part del motor de videojocs jMonkey Engine. És la classe encarregada de controlar totes les entrades, tant teclat, com ratolí com controladors de joc (gamePads, Joysticks,...). A partir d'ella, afegim les accions que volem executar quan passi un determinat esdeveniment. D'aquesta classe simplement ens fixem en la manera d'afegir accions i d'eliminar-les.

7.1.1.3.2. ThirdPersonHandler

Classe del motor de videojocs jMonkey Engine, que implementa un control del personatge molt complet imitant estils com els del Zelda Windwaker o Mario 64. El controlador que implementarem nosaltres heretarà d'aquesta classe, ja que

aprofitarem els controls que ens interessin i eliminarem els innecessaris, perquè aquest input també implementa una càmera, cosa que no ens interessa. Com podem veure té un sistema per tal d'escollir els botons que controlaran el jugador d'una manera molt senzilla.

7.1.1.3.3. EMdPHandler

Aquesta classe que hereta de **ThirdPersonHandler**, serà l'encarregada de fer un control de totes les entrades per a cada jugador. En ella definirem les accions necessàries i els esdeveniments que les executen.

Components de la classe:

- Atributs
  - **actionForward**: Acció que s'ha d'executar per anar endavant amb el jugador.
  - **actionBackwart**: Acció per fer el desplaçament endarrere.
  - **actionRight**: Per desplaçar-se cap a la dreta.
  - **actionLeft**: Per moure's cap a l'esquerra.
  - **actionDisparar**: És la acció que s'haurà d'executar quan el jugador vulgui llançar un paquet.
  - **actionPausar**: És la acció que s'executarà quan el jugador vulgui entrar al menú de pausa.
- Mètodes:
  - **EMdPHandler**: Constructor que inicialitza el controlador, així com assigna quins esdeveniments executen cada acció, i elimina les accions per defecte del **ThirdPersonHandler** que no interessen.
  - **update**: Actualitza els controladors.

7.1.1.3.4. KeyInputAction

KeyInputAction defineix una interfície per a la creació d'accions d'entrada. Aquestes accions poden correspondre a qualsevol esdeveniment definit per una tecla, o botó.

Com em dit anteriorment, la classe **InputHandler** tindrà un llistat de tots els **KeyInputActions**. Aquesta classe, és una classe abstracta i les accions heretaran d'aquesta.

7.1.1.3.5. EndavantEMdPAction

És la classe que defineix l'acció per moure endavant el jugador. Serà l'acció que s'executarà quan es defineixi l'esdeveniment a la classe **EMdPHandler**.

Components de la classe:

- Atributs:
  - **handler**: És una referència a la classe **EMdPHandler**, que controla les accions.
  - **personatge**: Conté el personatge que controla aquesta acció.
  - **rot**: La rotació d'aquest personatge.
- Mètodes:
  - **EndavantEMdPAction**: Constructor que inicialitza l'acció.
  - **performAction**: Mètode que s'executarà quan toqui reproduir l'acció. En aquest cas mourà el jugador cap endavant.

7.1.1.3.6. EndarreraEMdPAction

Aquesta classe serà igual que l'anterior, **EndavantEMdPAction**. Contindrà els mateixos atributs i els mateixos mètodes, amb l'única diferència que el mètode de reproduir l'acció, *performAction*, tindrà els canvis per tal de fer moure el jugador endarrere.

7.1.1.3.7. DretaEMdPAction

Aquesta classe també defineix una acció que s'ha de dur a terme en un determinat esdeveniment. Igual que les anteriors, contindrà els mateixos mètodes i atributs, amb l'única diferència que el mètode que executa quan succeeix l'esdeveniment, *performAction*, tindrà els canvis per tal de fer moure el jugador cap a la dreta.

7.1.1.3.8. EsquerraEMdPAction

La classe **EsquerraEMdPAction**, igual que les anteriors tres classes descrites, defineix l'acció que s'ha de dur a terme en un determinat esdeveniment. Igualment contindrà els mateixos mètodes i atributs, i l'única diferència està al mètode *performAction*, que canviarà per tal de que el jugador es mogui cap a l'esquerra.

7.1.1.3.9. DisparарEMdPAction

La classe **DisparарEMdPAction** defineix l'acció que es durà a terme en un determinat esdeveniment. Igual que les classes anteriors tindrà els mateixos atributs i mètodes, amb l'única diferència en el mètode *performAction*, que aquest enlloc de desplaçar el jugador, el que farà serà llançar un paquet cap a la posició a on mira i amb una determinada força segons quin personatge l'hagi llançat.

7.1.1.3.10. PausarEMdPAction

Aquesta classe defineix l'acció de pausar el joc i mostrar el menú de pausa. Igual que les anteriors tindrà dos mètodes, el constructor i el mètode *performAction*, que simplement farà una crida a la classe **GestorPantalla** per tal de desactivar el joc i posar la pausa.

7.2. Disseny de la persistència

Quan finalitza un procés, la memòria que aquest estava utilitzant queda alliberada, fent que es perdi tot el contingut. Si ens trobem en el cas que un objecte que ja ha estat creat en un procés determinat s'hagi de fer servir en un procés posterior, caldria emmagatzemar aquest objecte en un dispositiu d'emmagatzematge permanent per tal de poder ser utilitzat més tard. Aquest tipus d'objectes s'anomenen objectes persistents.

En aquest apartat que correspon al disseny de la persistència descriurem com hem decidit emmagatzemar les dades i gestionar-les, justificant aquelles decisions que hàgim pres.

El videojoc ha de permetre carregar diversos fitxers segons les necessitats bàsiques de l'aplicació. Aquests fitxers hauran d'emmagatzemar una configuració que pot ser modificada, una sèrie de dades dels escenaris i una altra amb la informació dels

personatges.

7.2.1.1. Emmagatzemament de dades

En l'aplicació hem decidit emmagatzemar les dades en fitxers. Tot i que en el disseny de la persistència, en cas de tractar-se d'emmagatzemar objectes, hauríem d'optar per una base de dades relacional, hem optat per a fer servir fitxers de configuració. Això ho hem fet perquè implementar una base de dades pel nombre de dades que gestionem seria totalment innecessari i altament costós. Per això, emmagatzemem les dades en fitxers, fent servir un format propi

Per tant, hem de guardar quatre tipus de dades dades diferents; configuració, personatges, escenaris i màximes puntuacions, crearem diferents fitxers, tots ells amb una extensió pròpia anomenada '.emdp'.

Dintre d'aquests fitxers, hem agafat les següents convencions:

- El símbol “#” ha estat escollit per a precedir els comentaris. Tota línia que comenci amb aquest símbol no serà interpretada per l'aplicació.
- El símbol “/” representa l'inici de la declaració de variables d'un objecte, anant sempre precedit per l'identificador d'aquest.
- Els valors que representin constants i no variables d'un objecte, són escrits amb el nom al davant en majúscules i sense espais seguits del símbol “=”. Per exemple, el valor ample s'escriurà com “AMPLE=”.

7.2.1.1.1. Fitxer de configuració: config.emdp

Aquest fitxer s'encarrega de guardar les dades bàsiques de configuració de l'aplicació, les quals es poden modificar des de la pantalla corresponent del videojoc. Totes aquestes dades son considerades constants.

Les dades que gestiona són:

- Alt i ample: resolució de la pantalla en format numèric d'alt i ample. Actualment l'aplicació està preparada per a configurar unes resolucions de 800x600 i de 1024x768, però s'han posat aquests valors per separat per a donar la possibilitat de possibles ampliacions.

- So: Un valor que indicarà si el so global de l'aplicació esta activat o no.
- Controls (de 1 fins a 4). Guarden el valor que representa el tipus de control que s'ha donat a cada jugador, sigui aquest el nom del controlador de joc que es fa servir, o el teclat.

7.2.1.1.2. Fitxer de dades d'escenaris: llistatEscenaris.emdp

En aquest fitxer emmagatzemem les dades dels escenaris que té el joc. No és modificable des de dins de l'aplicació, però és interessant que sigui un fitxer independent a les constants de codi per a poder afegir dinàmicament nous escenaris al joc, sense necessitat de tocar el codi font.

Entre les dades que conté aquest fitxer (del que podem veure un troç a mode d'exemple a la Figura 38) hi trobem:

- Id de l'escenari. Per a identificar-lo dintre l'aplicació.
- Nom de l'escenari.
- Número de bústies que conté aquest escenari, seguit de la posició d'aquestes bústies. Aquesta posició s'expressa amb un vector de tres dimensions.
- Número de personatges no jugadors. Igual que abans, primer trobem el nombre i després la seva posició en forma de vector.
- Furgonetes. Igual que els dos casos anteriors, tindrem emmagatzemat el número de furgonetes i la posició de la pantalla per la qual han d'arribar.
- I finalment, guardarem les posicions on han de néixer els personatges. Hi haurà quatre posicions que seran assignades a l'atzar per tal de no afavorir sempre al mateix jugador.

```
# Id de l'Escenari:
0
# Nom de l'Escenari:
Caribe
# Descripció de l'Escenari:
Lloc molt macu ple de pixa pins.
#BUSTIES:
#Num de bústies
2
# Posició de la bústia 0:
0,0.5f,8
# Posició de la bústia 1:
0,0.5f,-8
#PNJ:
# Quantitat de Pnj que hi hauran a l'Escenari
2
# Posició del PNJ 0:
2,0,0
# Posició del PNJ 1:
-2,0,0
#FURGONETES:
# Quantitat de Furgonetes que hi hauran a l'Escenari
2
# Posició de la Furgoneta 0:
9,1.5f,0
# Posició de la Furgoneta 0:
-8,1.5f,0
# JUGADOR:
# Número màxim de jugadors:
4
# Posició dels jugadors
1,0,1
1,0,-1
-1,0,1
-1,0,-1
```

Figura 38: Exemple de la configuració d'un escenari

7.2.1.1.3. Fitxer de dades dels personatges: llistatPersonatges.emdp

Aquest fitxer és molt similar a l'anterior, llistatEscenaris.emdp, i funciona seguint el mateix esquema, però conté dades diferents.

Les dades que conté són:

- Id del personatge, similar a l'identificador de l'escenari.
- Nom del personatge.

- Breu descripció del personatge.
- Nom del model 3D del personatge, així com el nom de la textura.
- Força del personatge, essent aquesta un valor entre zero i un. No guardem la velocitat ja que serà monòtonament decreixent a la força, seguint la fórmula *velocitat = 1-força*. Per exemple, un personatge amb una força de 0,42 tindrà una velocitat de 0,58.

7.2.1.1.4. Fitxers de registre de puntuacions: minimTemps.emdp i maximPaquets.emdp

Aquests dos fitxers segueixen la mateixa estructura, però un guardarà el registre de millors temps en mode per temps, i l'altre el de màxim número de paquets entrats en el mode per paquets.

Aquests fitxers guarden un registre dels millors deu del seu camp. En aquest registre s'hi troben les següents dades:

- Id del personatge.
- Nom del personatge. Tot i que el podríem extreure del fitxer llistatPersonatges.emdp, el guardem per a no haver de llegir de varis fitxers, ja que aquesta és una operació costosa.
- Puntuacio. En el cas del mode per paquets, es guarda el número de paquets. En el cas de per temps, es guarda el temps en segons.

8. Implementació

Una de les fases més importants dins de qualsevol projecte és la d'implementació. Un cop fet l'anàlisi i el disseny, passarem a implementar aquesta aplicació a través del llenguatge Java, el qual ens ha portat diferents conceptes útils per a millorar la distribució i eficiència de l'aplicació, com l'herència, les classes, etc.

En aquesta capítol parlarem sobre la implementació dels algoritmes més complexos i importants pel funcionament de l'aplicació final.

8.1. La classe ElMensajeroDePekin

Com hem explicat en els capítols anteriors, la nostre aplicació es tracta d'un videojoc que funcionarà a través d'estats. La classe ElMensajeroDePekin és la que conté el “main” de la nostre aplicació, i per tant és l'encarregada de preparar la gestió d'aquests estats, apart d'arrencar el joc pròpiament.

A continuació descriurem, com s'inicia el joc que gestionarà les actualitzacions i renderitzats de la nostre escena.

```
private static StandardGame _joc;
```

Figura 39: variable estàtica _joc de la classe StandardGame

En la Figura 39 podem veure la declaració de la variable estàtica de la nostre classe principal que representa l'execució del joc. Aquesta és de la classe StandardGame, explicada anteriorment, que té tot allò necessari per a gestionar el joc. Per a inicialitzar-la, li posarem uns valors de configuració per defecte i l'executarem.

```
_joc = new StandardGame("El Mensajero De Pekin");
valorsConfiguracioPerDefecte ();
carregaConfiguracio ();
_joc .start();

GestorPantalla.gestioPantalla();
```

Figura 40: Inici del “main” d'El Mensajero De Pekín.

En la figura 40 es pot veure com es crea el joc, si posen els valors de configuració per defecte, es carreguen i s'inicia el joc. Carregar els valors per defecte es veurà a la part de gestió de fitxers.

Un cop iniciat el joc, passem a crear les pantalles que tindrem, en forma d'estats, i a gestionar quines han de ser mostrades i quines no. D'això se n'ocupa la classe

GestioPantalla, que com podem veure a la Figura 40 s'inicia just començar el joc. El funcionament intern d'aquesta classe i els mètodes els veurem més endavant, a l'apartat de gestors.

8.2. Els estats

Dintre d'aquest apartat parlarem de la classe GbuiGameState, i d'aquelles que n'hereten, ja que son les encarregades de controlar els estats juntament amb l'ajuda de la classe GestorPantalla, que serà explicada més endavant.

8.2.1. GbuiGameState

```
public GbuiGameState(String name) {
            super(name);
            display = DisplaySystem.getDisplaySystem();
             BuiSystem.init(new PolledRootNode(Timer.getTimer(), input),
"/eMdP/estils/estil.bss");


             MouseInput.get().setCursorVisible(true);

            window = new BWindow(BuiSystem.getStyle(), new
AbsoluteLayout());
}
```

Figura 41: Constructor de la classe GbuiGameState

En la Figura 41 podem veure el constructor d'aquesta classe. Al crear cada estat del joc, inicialitzarem el BuiSystem, el cursor de ratolí i la finestra principal per a poder treballar amb la nostre interfície gràfica per als usuaris.

Dintre d'aquesta classe, el que és important de tenir en compte és com activarem i desactivarem, en general, les “pantalles” (quan parlem de “pantalles” ens referim a estats del joc, però els anomenem “pantalles” perquè és més gràfic).

```
public void activate() {
	// activa el main GameState
	super.setActive(true);

	// S'ocupa de mostrar la part del GBUI
	rootNode.attachChild(BuiSystem.getRootNode());
	 GameTaskQueueManager.getManager().update(new Callable<Object>() {
		public Object call() throws Exception {
			 BuiSystem.addWindow(window);
			window.center();
			return null;
		 }
	 });
	rootNode.updateRenderState();
}

public void deactivate() {
	// Si encara està activat, amaga la part del GBUI
	if (window.getRootNode() != null) {
		window.dismiss();
	}
	rootNode.detachChild(BuiSystem.getRootNode());

	// Desactivem la part del main GameState
	Figura 19:

	super.setActive(false);
}
```

Figura 42: Mètodes activate i deactivate de la classe GbuiGameState

Com es pot veure en la Figura 42, tenim un mètode per a cada una de les accions. L'important aquí és tenir en compte com treballa cada part. Per una part, tenim el **GameState**, que hem d'activar o desactivar per a controlar si es renderitza o no la informació d'aquell estat. D'això se n'ocupa la funció *setActive( );*, que es la que passa la informació al la part de **GameState** de les classes. Per altra banda, les altres instruccions d'aquestes dos funcions fan el mateix però amb la part d'interfície gràfica.

8.2.2. Estats per a iniciar partides

D'estats per a iniciar partides en tenim dos. Un per a les partides d'un jugador i l'altre per a les partides de dos jugadors. Com que el funcionament és molt similar, les explicarem conjuntes, puntualitzant, si cal, les diferencies entre elles en alguna funció concreta.

En general el que necessitarem aquí és escriure una sèrie de texts, crear els botons

per a seleccionar escenari i personatges amb imatges, uns botons per a seleccionar el mode, i finalment uns per a acceptar o cancel·lar, i en cas de acceptar, començar la partida recollint les dades entrades.

8.2.2.1. Iniciar i posar un títol

En el creador, el primer que fem és inicialitzar la mida de la pantalla que mostrarà texts, i crear el text d'inici, com es pot veure a la Figura 43.

```
public MenuIniciarPartidaUnJugador (String name)
{
	[...]

	window.setSize((int)(display.getWidth()),(int)
(display.getHeight()));

	int posicioY = display.getHeight()-GestorConstants.ALT_BOTO;

	text = new BLabel("Inicia partida d'un sol jugador");
	text.setPreferredSize(window.getWidth(), GestorConstants.ALT_BOTO);
	window.add(text, new Point(0, posicioY));

	[...]
}
```

Figura 43: Fragment del mètode MenuIniciarPartidaUnJugador

Aquí podem veure com donem la mida a la finestra mitjançant la funció *setSize( );* (que farem servir en tots els estats que mostrin informació per el GUI), i hi afegim el text de capçalera, a través del *window.add( )* i el **BLabel**.

8.2.2.2. Imatges i fletxes per a navegar-hi

Després, hem d'afegir les imatges i les fletxes per a triar escenari i personatge/s. El que fem és afegir un objecte contenidor al qual li apliquem una imatge de fons que modificarem amb les fletxes, les quals son dos elements botó. Es po veure com haurà de quedar a la Figura 44.

Figura 44: Imatge de com ha de quedar el selector de personatges i escenaris

```
 // Agafem el nom dels escenaris en format array d'Strings
String[] noms = getNomsPantalles();
// Agafem els escenaris de les pantalles
BImage[] imatges = getImatgesPantalles(noms);

// Comencem per l'escenariamb id=0
idPantalla = 0;

// Creem un "contenidor" que mostrarà  la imatge de fons
contenidor = new BContainer("Imatge pantalla");
contenidor.setBackground(BComponent.DEFAULT, new
ImageBackground(ImageBackgroundMode.SCALE_XY, imatges[idPantalla]));
contenidor.setPreferredSize(GestorConstants.PIXELS_IMATGE_PANTALLA,
GestorConstants.PIXELS_IMATGE_PANTALLA);
// Com que és un contenidor geèric, té habilitat el "hover", per tant el
deshabilitem
contenidor.setHoverEnabled(false);
window.add(contenidor, new Point(display.getWidth()/2-
GestorConstants.PIXELS_IMATGE_PANTALLA/2,posicioY));
```

Figura 45: Codi per a crear els contenidors per a imatges

Comencem amb les imatges. Com es veu a la Figura 45, el que fem és recuperar els noms de les imatges des d'un fitxer, i amb aquests agafar les imatges i posar-les en un vector de **BImage**. Llavors, creem un contenidor genèric **BContainer** que farem servir per a posar-hi la imatge de *background* per a mostrar-la. Fem un contenidor genèric, que podria ser qualsevol altre element, ja que és una classe més bàsica de GBUI que, per tant, hauria d'ocupar-nos menys càrrega per el sistema (tot i que amb aquestes classes tant simples tampoc ho notéssim). Finalment, l'afegim a l'objecte *window*. Amb el nom de l'escenari fem el mateix i l'afegim just a sota.

Després, el que hem de fer és afegir-hi els botons per a navegar-hi.

A la Figura 46 veiem com es crea un dels botons per a navegar entre les imatges que es podien veure a la Figura 44 (en aquest cas les imatges per a seleccionar escenari, el botó d'anar endarrere).

```
botoTmp.addListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
       String[] nomPantalles = getNomsPantalles();
       BImage[] imatges = getImatgesPantalles(nomPantalles);

       System.out.println("Pantalla anterior");
      idPantalla--;
      if(idPantalla<0)
            idPantalla = nomPantalles.length-1;
      window.getComponent(1).setBackground(BComponent.DEFAULT,
            new ImageBackground(ImageBackgroundMode.CENTER_XY,
      imatges[idPantalla]));

((BLabel)window.getComponent(2)).setText(nomPantalles[idPantalla]);

       }
});
```

Figura 46: Listener que conté el botó d'anar endarrere en el selector d'escenari

El que tenim és una variable de tipus **BButton** anomenada *botoTmp* a la qual l'hi afegim una acció, que haurà de ser disparada en cas de que es premi el botó, mitjançant la classe **ActionListener**. Aquí, el que fem és agafar la pantalla anterior, fent un control de que no sigui la primera, cas en el qual donaríem “la volta” i començaríem pel final, i la posem de *background* del component.

Per altra banda, tenim la funció que fa el contrari, posar l'escenari següent i en cas de ser el darrer, passar després al primer, com es pot veure en la Figura 47.

```
botoTmp.addListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
       System.out.println("Pantalla seguent");
       String[] nomPantalles = getNomsPantalles();
       BImage[] imatges = getImatgesPantalles(nomPantalles);
      idPantalla++;
      idPantalla=idPantalla%nomPantalles.length;
      window.getComponent(1).setBackground(BComponent.DEFAULT,
      new ImageBackground(ImageBackgroundMode.SCALE_XY,
imatges[idPantalla]));

((BLabel)window.getComponent(2)).setText(nomPantalles[idPantalla]);
       }
});
```

Figura 47: Listener que conté el botó d'anar endavant en el selector d'escenari

Un cop hem creat els menús selectors per a personatge i escenari, fets els dos de la mateixa manera però cada un amb el seu valor, creem els botons per a seleccionar mode.

8.2.2.3. Crear els botons del mode

Per al mode el que fem és crear uns botons **BToggleButton** que permetran, no tant sols seleccionar un valor, sinó que a més deixar-lo seleccionat, ja que son botons amb dos estats, activat i desactivat.

El que fem és crear dos botons i fer que quan un dels dos és seleccionat, desactivi l'altre botó, com es pot veure a la Figura 48. Aquest mateix procediment el farem dos vegades, intercanviant el valor del component que s'agafa de l'element *window* per a que la segona vegada sigui l'oposada a l'anterior.

```
mode = GestorConstants.MODE_TEMPS; // Mode = 1 vol dir paquets
// Creem el boto per temps
toggle = new BToggleButton("Per temps");
toggle .setPreferredSize(150,GestorConstants.ALT_BOTO);
toggle .setSelected(true);
// Afegim Listener que ens diu si esta seleccionat o no, ho apliquem al
boto i a mes actualitzem la variable mode
toggle .addListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
      boolean selected =
((BToggleButton)window.getComponent(10)).getState() == 3;
       ((BToggleButton)window.getComponent(11)).setSelected(selected);
      if(selected)
            mode = GestorConstants.MODE_PAQUETS;
      else
            mode = GestorConstants.MODE_TEMPS;
       }
});
// Afegim el boto a la finestra
window.add(toggle, new Point(display.getWidth()/2-160,posicioY));
```

Figura 48: Codi d'un boto toggle que al activar-se desactiva l'altre mode

8.2.2.4. Botons per a iniciar partida o retornar

Finalment, afegirem els botons per a entrar a la partida o bé retornar al menú principal.

A la Figura 49 podem veure com es creen aquests dos botons. Com en tots els casos, tenim un objecte declarat de manera “genèrica”, que reaprofitarem per a crear els dos botons. En el primer, el botó de tornar, veiem que l'únic que fem és un canvi de pantalla mitjançant el **GestorPantalla** i la funcio *canvia(estatActual, estatSeguent)* d'aquesta classe, que serà explicat més endavant.

El segon botó es crea molt semblant al primer, però aquest cop farem servir la funció *iniciarPartida()* del **GestorPantalla,** que el que farà serà crear un estat per a la partida nou, amb els valors que li passem per paràmetre. Aquesta funció també serà vista en detall a la secció de gestors.

```
// Botó de cancelar
boto = new BButton("Torna");
boto .setPreferredSize(GestorConstants.BOTO_200_PX,
GestorConstants.ALT_BOTO);
boto .addListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
      // Canviem estat actual per el menú d'inici
       GestorPantalla.canvia(3,0);
       }
});
window .add(boto, new Point(posicioX, 20));

 posicioX =
display.getWidth()/2+GestorConstants.ESPAI_ENTRE_ELEMENTS_GBUI;

// Botó per a iniciar partida
boto = new BButton("Juga!");
boto .setPreferredSize(GestorConstants.BOTO_200_PX,
GestorConstants.ALT_BOTO);
boto .addListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent event) {
      // Cridem al gestor de pantalla per a iniciar la partida i el canvi
d'estats
       GestorPantalla.iniciaPartida(idPlayer, idPantalla, mode);
       }
});
window.add(boto, new Point(posicioX, 20));
```

Figura 49: Codi per als botons de tornar enrere i jugar.

8.3. Gestors

En aquest apartat explicarem el funcionament dels gestors, una sèrie de classes que hem fet servir per a tasques no localitzades en un punt concret de l'execució, sinó que hi accedíem des de nombrosos punts i des de diferents classes.

8.3.1. El gestor de pantalla

El gestor de pantalla és una utilitat que fem servir per a crear els **GameStates**, afegir-los a una llista que servirà per a gestionar-los, i després intercanviar estats, sigui bé per a activar o desactivar amb mètodes bàsics, o sigui per tasques més completes com crear partides.

El **GestorPantalla** és cridat per primer cop al principi del joc, a on s'inicialitzen la majoria d'estats i s'afegeixen a la llista.

```
/**
* Metode que crida a la creació de la llista i afegeix els estats amb
* el activat o desactivat corresponent al GameStateManager
*/
public static void gestioPantalla()
{
	// Creem la llista
	inicialitzaPantalles();

	// Activem el primer estat
	llista.get(0).activate();
	for(int i=0;i<llista.size();i++)
	{
		if(i!=0) // El primer ja estÃ  activat, la resta els
desactivem
			llista.get(i).deactivate();
		// Afegim al GameStateManager
		 GameStateManager.getInstance().attachChild(llista.get(i));
	}
}
```

Figura 50: Mètode gestioPantalla

A la Figura 50 podem veure com s'inicialitzen les pantalles. El primer que fem és cridar el mètode *gestioPantalla*( ); que crea els estats bàsics necessaris i els afegeix a la nostre llista d'estats, com podem veure en la Figura 51. Després, activem el primer estat i desactivem la resta, i finalment els afegim tots a la llista d'estats del **GameStateManager**.

```
/**
* Metode que crea la llista d'estats i hi afegeix tots els estats que
* ja poden ser creats perquÃ¨ no modificarem
*/
public static void inicialitzaPantalles()
{
	// Creem una nova llista
	llista = new LinkedList<GbuiGameState>();

	// Afegim els estats
	llista.add(new MenuPrincipal("Menu Principal"));
	llista.add(new ThreeDGameState("Pantalla"));
	llista.add(new MenuConfiguracio("Menu Configuracio"));
	llista.add(new MenuIniciarPartidaUnJugador("Iniciar partida un
jugador"));
	llista.add(new MenuDePausa("Menu de Pausa"));
	llista.add(new MenuIniciarPartidaMultiJugador("Iniciar partida
multi-jugador"));
	llista.add(new Credits("Credits"));
	llista.add(new MaximesPuntuacions("Maximes puntuacions"));
}
```

Figura 51: Mètode inicialitzaPantalles

Aquesta classe també s'ocupa d'intercanviar estats cridant els mètodes de *activate( );* i *deactivate( );* com es pot veure a la Figura 52.

```
public static void canvia(int estatActual, int estatSeguent)
{
      llista.get(estatSeguent).activate();
      llista.get(estatActual).deactivate();
}
```

Figura 52: Mètode canvia

Finalment, de la classe **GestioPantalla** remarcarem el mètode que crea les partides. Bàsicament el que fem és cridar al constructor de partida amb els paràmetres que passa el menú que sigui que l'hagi cridat, i activar-lo afegint-lo, a més, al **GameStateManager**. A la Figura 53 podem veure el funcionament del mètode.

```
/**
* Metode que crea i inicialitza una partida multijugador
* @param idPlayers vector d'enters amb els id's dels personatges (-1 si
no juga)
* @param idEscena enter amb l'id de l'escena
* @param mode enter amb el mod (veure constants per als modes)
*/
public static void iniciaPartida(int[] idPlayers, int idEscena, int mode)
{
      // Creem la nova partida i l'afegim al GameState
      llista.add(new Partida(idPlayers, idEscena, mode));
      GameStateManager.getInstance().attachChild(llista.getLast());
      // Activem la Ãºltima partida i desactivem el menÃº corresponent
      llista.getLast().activate();
      llista.get(5).deactivate();
}
```

Figura 53: Mètode iniciaPartida per a mode multijugador

8.3.2. Càrrega de models i textures: el gestor de models

Per a gestionar la càrrega de models i textures, ja que hem treballat amb diferents fitxers (.3ds i *-jme.xml), hem creat una classe per a gestionar-ho.

Primer explicarem l'importador de fitxers *-jme.xml. Aquest és un format especific del nostre motor, el qual hi hem treballat gràcies al HottBj, un plugin per a Blender que permet exportar la feina feta en aquest format.

Per a importar aquest tipus de models, només hem de crear un XMLImporter, ja que és el format per defecte del jMonkey Engine. En la Figura 54 podem veure la implementació.

```
public static Node getJMEXML(String model)
{
      URL modelToLoad =
GestorModels.class.getClassLoader().getResource(model);

      try {
            XMLImporter importer = new XMLImporter();
            Node result = (Node)importer.load(modelToLoad);
            logger.info("ModelImporter:Model *-jme.xml carregat amb
éxit!");
            return result;
       } catch (IOException e) {
             System.out.println("ModelImporter::ERROR al carregar el
model: "+model);
            return null;
       }
}
```

Figura 54: Implementació del mètode getJMEXML

Bàsicament, el que fa és carregar un model amb la classe **XMLImporter** i afegir-lo a un node, que després farem servir d'escenari.

Per altra banda, per a afegir una textura a un model amb aquest mateix format, tenim una funció que engloba aquesta operació.

El més important d'aquest mètode es pot veure a la Figura 55 . Bàsicament, creem un **TextureState** que contindrà la textura, l'habilitem, carreguem el fitxer que li hem passat de textura, i finalment l'afegim al personatge.

```
TextureState ts = display.getRenderer().createTextureState();
ts.setEnabled(true);
ts.setTexture(
TextureManager.loadTexture(
       GestorModels.class.getClassLoader().getResource(fitxerTextura),
       Texture.MinificationFilter.Trilinear,
       Texture.MagnificationFilter.Bilinear)
);
personatge.setRenderState(ts);
```

Figura 55: Creació de la textura i afegiment al personatge

Finalment parlarem del mètode que s'encarrega de carregar models en format .3ds. Aquest mètode simplement carrega un model i l'afegeix a un node, igual que hem fet amb l'anterior mètode de carrega de models. Bàsicament, el que fa aquest mètode és agafar el model amb un objecte **ByteArrayOutputStream**, que és de la manera necessària per a fitxers .3ds. En la Figura 56 es pot veure el funcionament de la

càrrega del model.

```
MaxToJme C1 = new MaxToJme();
ByteArrayOutputStream BO = new ByteArrayOutputStream();
C1.convert(new BufferedInputStream(modelToLoad.openStream()), BO);
Node resultat = (Node)BinaryImporter.getInstance().load(new
ByteArrayInputStream(BO.toByteArray()));
resultat.getLocalRotation().fromAngles(-FastMath.HALF_PI, FastMath.PI,
0);
```

Figura 56: Càrrega d'un model en 3DS

8.3.3. Lectura i escriptura a fitxers: GestorFitxers

Per tal de gestionar l'accés a fitxers, hem implementat la classe **GestorFitxers.**

8.4. Les classes Bàsiques

En aquest apartat explicarem la implementació d'aquelles classes que les em classificat com a classes bàsiques, ja que contenen els elements principals del videojoc. Aquests elements són:

- Personatge.
- PersonatgeNJ.
- Bústia.
- Paquet.
- Furgoneta de correos.
- Escenari.
- Registre de Puntuacions.

A continuació explicarem la implementació de cada una d'aquestes classes, enumerant simplement els mètodes més importants.

8.4.1. Personatge

Alhora de moure els personatges jugadors i no jugadors per l'escenari s'ha de tenir en compte que aquests no travessin les parets ni els altres obstacles col·locats a mig de la pantalla. Això s'ha de fer, ja que si no s'implementa un control de detecció de

col·lisions, al ser tots els objectes malles, es poden interposar unes a dins de les altres. Per resoldre aquest problema hi han varies opcions, nosaltres escollirem utilitzar el motor de física Physics. D'aquesta manera serà el motor encarregat de controlar les col·lisions dels personatges per la pantalla.

Per tal d'implementar aquest sistema, utilitzarem uns nodes que seran els que gestionarà el motor de física.

Tal com es veu a la Figura 57 podem observar que hi hauran dos nodes importants, un que serà una càpsula i representarà el cos, i un altre que serà una esfera i representarà els peus del personatge. Utilitzarem una esfera pels peus ja que aquesta gira independentment del model, i no provoca cap mena de fricció. A part, ajuntarem l'esfera amb la càpsula mitjançant una força és mantindrà alineada amb la normal del terra.

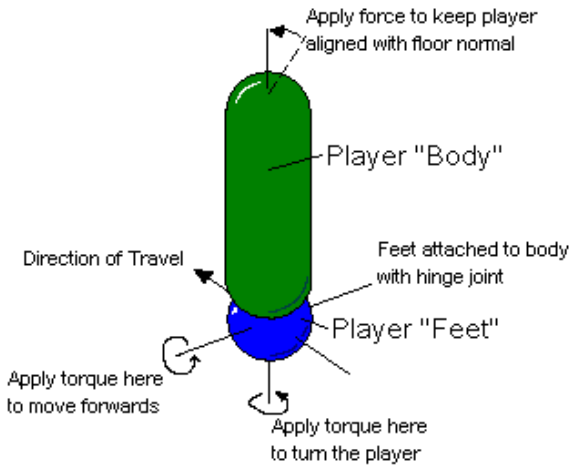


Figura 57: Imatge de la combinació de dos Bounding Box per a gestionar les col·lisions d'un personatge

A l'hora de carregar el model del personatge, haurem de controlar l'escala d'aquest, ja que si el model és molt gran tindrem una simulació molt lenta. Per altra banda, si el model és molt petit podrem comprovar que, a vegades, aquest travessa la meitat dels objectes.

Per implementar aquest esquema, el constructor, primer de tot, carrega el model del **Personatge** o del **PersonatgeNJ** a través del mètode *getJMEXML* de la classe **GestorModels**, explicada anteriorment. Un cop el model carregat, ja tenim les propietats gràfiques del **Personatge**, com l'alçada i l'amplada. Amb aquests valor

podrem construir els nodes de física com es veu a la Figura 58.

Figura 58: Nodes de física d'un model

Per crear l'esfera que simula els peus, agafarem com a valor de radi, l'amplada de la caixa envolupant del personatge, com és pot veure a la Figura 59. A més també assignarem com a material de l'esfera el granit, a causa que té unes propietats físiques que són les que ens interessen: rigidesa, superfície rugosa i una alta densitat.

```
float cRadius = bbox.xExtent < bbox.zExtent ? bbox.zExtent :
bbox.xExtent;
float cHeight = bbox.yExtent * 2f;

//Creem l'esfera que simularà  els peus del personatge
//  1. Creem el node dynamic de l'esfera
sphere = space.createDynamicNode();
sphere .setName("Sphere");
//  2. Creem una esfera física que serà  filla de l'esfera
PhysicsSphere s = sphere.createSphere("BodySphere");
sphere .attachChild(s);
//  3. Assignem a l'esfera física el radi agafat anteriorment
s.setLocalScale(cRadius);
//  4. Assignem el material, per tal de que és mogui de una
//determinada manera
sphere .setMaterial(Material.GRANITE);
sphere.computeMass();
```

Figura 59: Codi per a crear la caixa envolupant dels peus del personatge

Després de crear l'esfera, seguim creant la càpsula que serà la que representarà el tronc del **Personatge**. Per crear aquesta càpsula agafarem el mateix radi que per crear l'esfera, i l'alçada l'agafarem de la caixa envolupant. A més també haurem de

girar la càpsula per tal de que ens quedi vertical, i rotar-la cap allà a on mira el **Personatge,** tal i com es veu a la Figura 60**.**

```
//  1. Creem el node dinàmic de la càpsula
capsula = space.createDynamicNode();
capsula .setName("Capsula");
//  2. Creem una càpsula física que serà  filla del node dinàmic capsula
PhysicsCapsule c = capsula.createCapsule("CapsulaCos");
capsula .attachChild(c);
//  3. Assignem el tamany d'aquesta càpsula
c.setLocalScale(new Vector3f(cRadius,cHeight,cRadius));
//  4. I la posició a on estarà  col·locada
c.setLocalTranslation(0, cRadius*4, 0);
//  5. Finalment agafem la rotació cap a on ha de mirar
Quaternion rot = new Quaternion();
rot.fromAngleAxis(FastMath.HALF_PI, Vector3f.UNIT_X);
c.setLocalRotation(rot);
capsula .computeMass();
//  6. Assignem el model a la càpsula, perquè és moguin junts
capsula .attachChild(result);
//  7. Girem el model perquè quedi de peus
result.getLocalRotation().fromAngleAxis(FastMath.HALF_PI, new Vector3f(-
      1,0,0));
```

Figura 60: Codi per a crear i rotar la càpsula que envolta al personatge

A continuació de la part física, el constructor també s'encarregarà de carregar la part gràfica del **Personatge**, així que, mitjançant el mètode *afegirTexturaXML* de la classe **GestorModels**, carregarà les textures. Finalment el constructor inicialitzarà les animacions del **Personatge**, carregant l'animació d'esperar, com es pot veure a la Figura 61.

```
if(superBone.getControllerCount() > 0){
      animacio = (AnimationController)superBone.getController(0);
      if(animacio.hasAnimation("Esperar")){
            animacio.setActiveAnimation("Esperar");
            animacio.setActive(true);
       }
}
```

Figura 61: Codi per a carregar l'animació d'esperar

Un altre mètode important de la classe **Personatge** és el que assigna els paquets un cop ha col·lisionat contra una **FurgonetaDeCorreos.** Aquest mètode, que podem veure a la Figura 62, crea tants paquets com li indiquem mitjançant el **GestorConstants**, i comprova que no s'hagin creat dos paquets del mateix color. Un cop creats, els afegeix a una **LinkedList**, per finalment cridar al mètode *actualitzarPaquets.* Aquest agafarà l'últim paquet del llistat i l'assignarà al

Personatge, així serà present durant el joc. Quan el **Personatge** llanci un **Paquet**, tornarem a cridar a aquest mètode, per tal de tenir sempre un **Paquet** a la mà.

```
public void carregar(PhysicsSpace espaiFisic, int numPaquets){
      int ultimColor = -1;
      for (int l = 0; l < numPaquets; l++)
       {
      //Creem un num aleatori per tal de escollir el color del paquet
            int color =
                  (int)Math.round((Math.random()*GestorConstants
                        .NUM_COLORS));
            if(ultimColor != -1)
             {
                  if(color == ultimColor)
                        l--;
                  else
                   {
                        Paquet paq = new Paquet("Paquet"+color,color
                              ,espaiFisic.createDynamicNode());
                       paquets.addLast(paq);
                        ultimColor = color;
                   }
             }
            else
             {
                  //assignarPaquet(espaiFisic, color);
                   Paquet paq = new Paquet("Paquet"+color,color
                        ,espaiFisic.createDynamicNode());
                  paquets.addLast(paq);
                   ultimColor = color;
             }
      }
      actualitzarPaquets();
}
```

Figura 62: Mètode carregar

A part d'aquests mètodes, i de totes les funcions per agafar/assignar atributs de la classe, també tenim tots els mètodes per tal de controlar les animacions. Aquests mètodes, com és veu a l'exemple de la Figura 75, el que fan és activar/desactivar una animació, i en cas d'activar-la, assignar-li una velocitat. En el cas específic de llançar un **Paquet,** tenim el mètode *haAcabatDeTirar* que retorna un booleà indicant-nos si l'animació del **Personatge** ha acabat o no.

8.4.2. Classe PersonatgeNJ

Tal i com s'ha vist en el capítol de disseny, la classe **PersonatgeNJ** hereta de la classe Personatge per tant també heretarà els seus mètodes així com el sistema de mobilitat (tant per la part de les animacions com per la part de la física). La diferencia

principal entre un **Personatge** i un **PersonatgeNJ** és que els **PersonatgeNJ** han de tenir una part d'intel·ligència artificial per tal que aquest persegueixi al jugador fins a atrapar-lo.

Per implementar la intel·ligència artificial, farem servir un mètode que actualitzarà la posició del **PersonatgeNJ**, depenent d'on es trobin la resta de jugadors. Com es veu a la Figura 63, el mètode *actualitzar* primer de tot actualitza la força que se li aplica a la càpsula (explicat a l'apartat anterior de **Personatge**) i seguidament comprovem que no estigui bloquejat. Si no està bloquejat, buscarem els personatges. Primer de tot llançem un raig, a on es guardaran totes les col·lisions, comprovarem que no hi hagi cap jugador a davant. En cas positiu, avançarem la posició del **PersonatgeNJ**. Altrament cridarem al mètode *buscarAlVoltant,* explicat a continuació.

```
// Actualitzem la força per mantenir dempeus el PNJ
update(Timer.getTimer().getTimePerFrame());
// Comprovem que no estigui bloquejat
if(!this.estaBloquejat(temps)){
      Vector3f origen = new Vector3f(this.getLocalTranslation());
      origen.setY(0.6f);
      // LLancem Raig endevant per veure si veiem algu
      PickResults resultats = llançarRaig(origen,
            this.getLocalRotation().getRotationColumn(2, new
                  Vector3f()));
      if(hiHaJugadorDavant(resultats,temps)){
             Vector3f loc = this.getLocalTranslation();
             loc.addLocal(capOnMira()).multLocal(getVelocitat());
             this.setLocalTranslation(loc);
      }else{
            this.setLocalRotation(buscarAlVoltant(origen,temps));
      }
}
```

Figura 63: Fragment del mètode buscarAlVoltant

El mètode *buscarAlVoltant*, simplement llença rajos en totes les direccions al voltant del paràmetre *origen*, buscant un **Personatge.** Per tal de saber si un objecte que travessa el raig és un **Personatge** o no, hem implementat el mètode *hiHaJugadorDavant*, que passant-l'hi el resultat de llançar d'un raig, fa la comprovació, tal com es pot veure a la Figura 64.

```
private boolean hiHaJugadorDavant( PickResults results,float temps){

      boolean trobat = false;

      if(results.getNumber() > 1) {
      // Agafem la col·lisio 1, perqué la 0 conté el model del pnj
            PickData closest = results.getPickData(1);
           if(closest.getDistance()<visio &&
                 esFillDUnPersonatge(closest.getTargetMesh(),temps)){

                 trobat=true;
       }
      return trobat;
}
```

Figura 64: Mètode hiHaJugadorDavant

8.4.3. Classe FurgonetaDeCorreos

El mètode més important de la classe **FurgonetaDeCorreos** és l'*actualitzar.* Aquest comprova a quina posició està la representació gràfica de la furgoneta, executant unes funcions o unes altres, tal i com es pot veure a la Figura 65. Segons la seva posició i si té carregaments, aquesta es mourà en sentit a la zona de descàrrega, o es mourà en sentit a la zona d'espera, o simplement estarà esperant.

```
public void actualitzar(float tIS){
      if(this.carregaments > 0 &&
            (this.getLocalTranslation().distance(getPuntArribada())>1)){
         if(!estaBloquejat(tIS))this.anarA(0,tIS); //Va a Descarregar
      }else if(this.carregaments <1 &&
            (this.getLocalTranslation().distance(getPuntFinal())>1)){
         if(!estaBloquejat(tIS))this.anarA(1,tIS); //Va a Carregar
      }else if(this.carregaments==0 &&
            (this.getLocalTranslation().distance(getPuntFinal())<1)){
         this.bloquejar(tIS, tIS+5);
         //Carreguem, i esperem 5s perqué es dirigeixi cap a la partida
         setCarregaments(1);
      }
}
```

Figura 65: Mètode actualitzar de la classe FurgonetaDeCorreos

8.4.4. Escenari

En aquest apartat explicarem el mètode més important de la classe **Escenari**, el que carrega l'escenari gràfic. Per tal de controlar les col·lisions del **Personatge** mitjançant el motor de física Physics, aquest té dues opcions. La primera, el motor

crea automàticament les caixes envolupants de tots els objectes, arribant a un punt on el proces de calcular les col·lisions és massa costós. La segona opció: crear manualment caixes físiques al voltant dels objectes a on es poden produir les col·lisions. Nosaltres escollim la segona opció, ja que alliberem al motor de física controlar col·lisions que no ens interessen.

Per dur-ho a la pràctica, al moment de crear l'escenari (parlant gràficament), haurem de col·locar uns cubs allà a on volem que hi hagin col·lisions i anomenar-los “o_invisible”. D'aquesta manera el mètode *carregarModel* de la classe **Escenari,** com es veu a la Figura 79, crearà caixes de la mida del objecte. A més, també farà que els cubs amb aquests noms siguin invisibles.

```
Node mentre = GestorModels.getEscenari(direccio,textura);
entre.setModelBound(new BoundingBox());
mentre.updateModelBound();
mentre.updateWorldBound();

for(int i = 0; i< mentre.getChildren().size(); i++)
{
       if(mentre.getChild(i).getName().contains("o_invis"))
       {

            PhysicsBox obstacle = nodeEstatic.createBox("obstacle"+i);
            mentre.getChild(i).updateWorldBound();
            BoundingBox bbox = (BoundingBox)
                  mentre.getChild(i).getWorldBound();

            obstacle.getLocalScale().set( bbox.xExtent * 2, bbox.yExtent
                  * 2, bbox.zExtent * 2 );

            obstacle.getLocalTranslation().set( bbox.getCenter() );

            this.nodeEstatic.attachChild(obstacle);

            // Els tornem invisibles
            ((Node)mentre.getChild(i)).setCullHint(CullHint.Always);
      ((Node)mentre.getChild(i)).setRenderQueueMode(Renderer.QUEUE_SKIP);
       }
}
this.nodeEstatic.attachChild(mentre);
this.nodeEstatic.getLocalRotation().fromAngleAxis(-FastMath.HALF_PI, new
      Vector3f(0,1,0));
```

Figura 66: Codi per a crear caixes al voltant dels objectes o_invisible

8.5. La classe partida

La classe **Partida**, és la que controla la partida en si. És l'encarregada de inicialitzar

el valor dels personatges, pnjs, bústies, controls d'entrada, ... així com actualitzar aquests valors i posicions. Té dos constructors, un per a crear la **Partida** individual i un altre per a la multijugador. Com hem vist a l'apartat de Disseny, la diferència entre els dos constructors és que, en individual, rep un enter amb l'id del **Personatge,** i en multijugador rep una cadena d'ids de **Personatges.** Per tant el constructor de la partida individual, el que farà serà crear una cadena amb l'id del **Personatge,** ja que durant el bucle de joc (Game loop) sempre treballarem amb cadenes. Els dos mètodes cridaran a la funció *inicialitzemElements*, el qual crearà tots els elements necessaris de la **Partida.**

Primer de tot, aquest mètode crearà l'element més important, que serà l'**Escenari**. Mitjançant el mètode *carregarEscenari* de la classe **GestorFitxers** llegirem un fitxer que contindrà tota la informació referent a un **Escenari** (com hem explicat anteriorment). Aquest fitxer *serà* el que definirà quantes bústies, pnjs, jugadors i furgonetes hi ha a la pantalla.

Un cop creat l'**Escenari**, crearem els jugadors d'aquest, passades per paràmetre les ids. Com podem veure a la Figura 80, farem un recorregut per totes les ids dels **Personatges**, cridant al mètode *carregarPersonatge* de la classe **GestorFitxers**, que, com hem explicat anteriorment ens crearà un **Personatge**, amb les característiques escollides segons l'identificador. Seguidament assignarem al **Personatge** l'id del jugador que el controlarà, el col·locarem al punt que indica l'**Escenari** i farem que miri al centre de la pantalla. A més, també l'afegirem a la **LinkedList,** que contindrà un llistat dels jugadors i farem que l'**Escenari** sigui el seu pare.

```
Personatge a;
for(int i=0;i<idsPersonatges.length;i++){
      a = GestorFitxers.carregarPersonatge(idsPersonatges[i],
            getPhysicsSpace());
      a.setIdJugador(i);
      a.setLocalTranslation(escenari.getPosJugadors().get(i));
      a.mirarAlCentre();
      escenari.attachChild(a);
      jugadors.add(a);
}
```

Figura 67: Part del codi per a crear els personatges a l'inici d'una partida

Després de crear els jugadors, crearem de manera semblant els **PersonatgeNJ**, les **Bústies** i les **FurgonetesDeCorreos.**

Un cop els elements bàsics del joc estan creats, continuem creant les llums. La llum de la **Partida** serà una llum ambient d'un color blanc. Aquesta estarà col·locada a un punt elevat de l'**Escenari** per tal de que il·lumini tot el contingut. Seguirem col·locant la càmera fixa a un punt concret i elevat. Una altra qüestió important és la creació dels controls d'entrada. Tal i com és veu a la Figura 67, tindrem una cadena de controladors d'entrada, a on cadascuna d'aquestes controlarà a un Personatge. A més, llegirem del fitxer de configuració per a saber quines seran les tecles a fer servir per cada jugador.

```
inputeta = new EMdPHandler[numInputs];
for(int i=0;i<numInputs;i++){
    HashMap<String, Object> handlerProps = new HashMap<String,Object>();
    handlerProps.put(EMdPHandler.PROP_CONTROL_ENTRADA,
            GestorFitxers.controls(i);
    inputeta[i] = new EmdPHandler(jugadors.get(i),cam,handlerProps);
}
```

Figura 68: Creació dels controls d'entrada

Finalment, inicialitzarem la funció de *Callback,* que serà l'encarregada de controlar la física del nostre joc, tal i com expliquem al següent apartat (Física).

A més del constructor, l'altre mètode important de la Partida és el mètode *update*. Aquest és el que gestiona el bucle de joc (Game Loop), cridant als mètodes actualitzar dels **PersonatgesNJ**, de les **FurgonetesDeCorreos** i dels controladors d'entrada per cada jugador.

8.6. Física

En aquest apartat parlarem de com està implementada la física en el videojoc. Tal i com em explicat anteriorment en el capítol de disseny, tenim una serie de col·lisions que hem de controlar per tal de poder executar unes funcions determinades segons quina col·lisió hagi succeït. El motor de física Physics, integrat a dins del jME, serà l'encarregat de controlar-ho.

Per tal de indicar-li al Physics que ha de fer quan succeeixi una col·lisió, hi han les funcions de *CallBack*. Mitjançant aquestes funcions, indiquem al motor el que volem

que passi quan xoquin uns determinats objectes. Per crear aquestes funcions hem creat la classe **ImplCallBack,** que serà la que contindrà aquestes funcions.

Amb aquesta classe controlarem les col·lisions dels nodes dinàmics. Per tant, bàsicament, controlarem el que passa amb els personatges**,** amb els personatges no jugadors i amb els paquets. Aquestes col·lisions estan enumerades al capítol de disseny.

```
ContactCallback myCallBack = new ContactCallback() {
    public boolean adjustContact( PendingContact c ) {
       // Comenem amb les col·lisions que hi intervé un Paquet
      if(c.getNode1().getName().contains("Paquet") ||
           c.getNode2().getName().contains("Paquet")){

                    ...

      // Seguim amb les col·lisions que hi intervé un Personatge
      }else if(c.getNode1().getName().contains("Personatge") ||
                 c.getNode2().getName().contains("Personatge")){

                     ...

     }
```

Figura 69: Divisió global de les col·lisions segons provinguin d'un paquet o d'un personatge

Tal i com es veu a la Figura 69, podem comprovar com en aquesta funció dividim en dos grans blocs el codi. Per una part, les col·lisions que hi intervé un **Paquet**, i per altre part les que hi intervé un Personatge. Seguidament, a dins de cada bloc comprovem quin és l'altre element que intervé a la col·lisió.

8.6.1. Col·lisió d'un paquet contra un personatge o un PNJ

Com es pot veure a la Figura 70, aquesta és la funció que s'executa quan xoquen un paquet contra un personatge o un personatgeNJ. Aquesta funció, primer de tot, comprova que el personatge no estigui en un estat immune, ja que si aquest ho està, no li afectarien els cops de paquet. Seguidament reprodueix un so 3D, a la posició que nosaltres li indiquem, fent més real la col·lisió. A continuació entra al bucle a on eliminem dos dels paquets que porti aquests personatge. En cas que la col·lisió fos de un paquet contra un personatgeNJ, aquest no portaria cap paquet i no entraria al bucle. Actualitzem el valor per pantalla que representa el nombre de paquets que té el jugador, i finalment el bloquegem dos segons**.**

```
//Comprovem que el personatge no sigui immune
if (!p.isImmune(Timer.getTimer().getTimeInSeconds())){
      //Reproduim un so de col·lisió, amb idColisio igual a 1
      GestorMusica.reproduirSoColisio(1,p.getLocalTranslation())
     //Eliminem 2 paquets del jugador
     int cont=0;
     // En cas de ser un PNJ no tindrà  paquet i no entrerà  al bucle
     while(p.getNumPaquets() > 1 && cont < 2){
         //Eliminem l'últim paquet del personatge
         p.getPaquets().removeLast();
         Cont++;
         //Mostrem el número de paquets que té el personatge
         ((Blabel)GestorPantalla.getLlista().getLast().getFinestra()
            .getComponent(4*p.getIdJugador()+4)).setText(""
                  +p.getNumPaquets());
      }
      //Esperem 2 segons sense poder moure el personatge o el PNJ
      p.bloquejar(Timer.getTimer().getTimeInSeconds()+2);

}
```

Figura 70: Codi que s'executa quan xoquen un paquet i un personatge

8.6.2. Col·lisió d'un paquet contra una bústia del mateix color

```
paq.setActive(false);
paq.getLocalTranslation().set(-20,-8,-20);
Personatge p = (Personatge)((Escenari)rootNode.getChild(0))
      .getChild(paq.getIdPersonatge()+1);
p.paquetEntrat();
//Música que el paquet ha entrat a la bústia correcte!
GestorMusica.reproduirCritColisio(3,p.getLocalTranslation());
paq.getNodeFisic().delete();
paq.getParent().detachChild(paq);
((Blabel)GestorPantalla.getLlista().getLast().getFinestra()
      .getComponent(4*p.getIdJugador()
            +3)).setText(""+p.quantsPaquetsHaEntrat());
```

Figura 71: Col·lisió d'un paquet contra una bústia del mateix color

La Figura 71 mostra el codi que s'executarà en cas de succeir la col·lisió de un paquet contra una bústia del mateix color. Primer de tot desactivarem la física del paquet, ja que un cop a dins de la bústia hem d'eliminar-lo. Seguidament incrementarem el comptador de paquets del jugador i reproduirem un so per tal d'indicar que hem encertat la bústia. Finalment, eliminarem el paquet, i actualitzarem el comptador que mostra el nombre de paquets entrats per pantalla.

8.6.3. Col·lisió d'un paquet contra una bústia de colors diferents

```
GestorMusica.reproduirSoColisio(4,bus.getLocalTranslation());
Vector3f dirForsa = new Vector3f();
Personatge p = (Personatge)(rootNode.getChild(paq.getIdPersonatge()+1));
dirForsa.set(p.getLocalTranslation());
dirForsa.subtractLocal(bus.getLocalTranslation());
paq.addForce(dirForsa);
paq.setHaEstatLlançat(true);
```

Figura 72: Codi que s'executa al xocar un paquet contra una bústia de diferent color

A la Figura 72 podem veure el que executarem quan hi hagi una col·lisió de un paquet contra una bústia d'un color diferent. Tal i com hem explicat anteriorment, quan un paquet col·lisiona contra una bústia d'un color diferent, aquesta ha de llançar el paquet amb força per tal de tocar el personatge que l'ha llançat. Per tant, primer de tot reproduirem un so de col·lisió tridimensional, a la posició de la Bústia, i seguidament agafarem la posició del personatge. A continuació mitjançant la posició del personatge i la de la bústia, calcularem el vector de la força que aplicarem sobre el paquet de la següent manera: Força = (Posició del personatge – Posició de la bústia)*10.

Un cop explicades les funcions que executarem quan un dels dos elements de la col·lisió sigui un paquet, veurem les funcions quan un dels dos elements col·lisionats sigui un personatge. Hi hauran dos tipus de col·lisions: un personatge contra un personatge no jugador i un personatge contra una furgoneta de correus.

8.6.4. Col·lisió d'un personatge amb un personatgeNJ

El següent codi de la Figura 73, mostra el que executarem en cas del primer tipus de col·lisió de un personatge. Un personatge contra un personatge no jugador. Com podem veure, el primer que fem és comprovar que el personatge no estigui bloquejat. Després comprovem que el personatge no jugador tampoc no estigui bloquejat. Si cap dels dos ho està, els bloquejarem tots dos, ja que significarà que el personatge no jugador ha atrapat al personatge. En cas contrari, el personatge que no estigués bloquejat, prosseguiria el seu camí.

```
/* Comprovem que el Personatge no està bloquejat */
if (!p.estaBloquejat(Timer.getTimer().getTimeInSeconds())) {
/* Si el PNJ està  bloquejat no passa res per col·lisionar amb ell */
      if (!pNJ.estaBloquejat(Timer.getTimer().getTimeInSeconds())) {
            /* Si cap dels dos està  bloquejat, bloquegem els dos */
            p.bloquejar(Timer.getTimer().getTimeInSeconds() + 8);
            pNJ.bloquejar(Timer.getTimer().getTimeInSeconds() + 14);
      }
}
```

Figura 73: Comprovació de que els personatges no estiguin bloquejats

8.6.5. Col·lisió d'un personatge amb una furgoneta

Finalment, explicarem l'últim tipus de col·lisió: un personatge contra una furgoneta de correus. Quan succeeix aquesta col·lisió, executarem el codi que mostra la Figura 74. Com podem veure el primer que fem és comprovar que el personatge no tingui paquets, i que la furgoneta de correus pugui descarregar. Seguidament, si el personatge no està bloquejat, reproduirem un so de col·lisió per tal de donar realisme a la col·lisió. A més a més, posarem el personatge en estat immune i el bloquejarem. A continuació cridarem al mètode *carregar* de la classe personatge, a on, com hem explicat anteriorment, assignarem els paquets al personatge. Finalment li indicarem a la massa detallat fent molt costós furgoneta de correus que ha descarregat, i actualitzarem el valor que ens indica per pantalla, el nombre de paquets que té el jugador.

```
if (p.getNumPaquets() < 1 && furgo.potDescarregar()) {
/* Comprovem que el jugador no estigui bolquejat */
      if (!p.estaBloquejat(Timer.getTimer().getTimeInSeconds())) {
            GestorMusica.reproduirSoColisio(6,furgonetes.get(k)
                  .getLocalTranslation());
            p.setImmune(Timer.getTimer().getTimeInSeconds() + 6);
            p.bloquejar(Timer.getTimer().getTimeInSeconds() + 3.5f);
            p.carregar(espaiFisic,GestorConstants
                  .NUM_PAQUETS_CARREGAMENT);
          /* Indiquem a la furgoneta que ha fet una descàrrega */
          furgo.unCarregamentFet(Timer.getTimer().getTimeInSeconds());
          ((Blabel)GestorPantalla.getLlista().getLast().getFinestra()
                .getComponent(4*p.getIdJugador()+4))
                      .setText(""+p.getNumPaquets());
      }
}
```

Figura 74: Col·lisió de furgoneta amb personatge

8.7. Control d'entrades

En aquest apartat descriurem com implementarem els mètodes per tal que un usuari pugui moure un personatge del videojoc. Per poder-lo moure, podrà escollir entre teclat o controladors de joc (GamePads, Joysticks,.etc.).

Com hem dit anteriorment, la classe EMdPHandler, serà l'encarregada de fer aquest control de les entrades, i executar les funcions que siguin necessàries. El funcionament és seguint una serie d'accions que nosaltres definirem, i indicant-li a la classe principal EMdPHandler quina acció ha d'executar quan succeeixi un determinat esdeveniment.

8.7.1. Classe EMdPHandler

```
super .updateProperties(props);
KeyBindingManager keyboard = KeyBindingManager.getKeyBindingManager();

keyboard.set(PROP_KEY_F10, getIntProp(props, PROP_KEY_F10,
       KeyInput.KEY_F10));
```

Figura 75: Actualització de les propietats i de les tecles de control

Tal i com podem veure a la Figura 75, en el constructor del EMdPHandler, el primer que farem a part d'inicialitzar els atributs de la classe, serà actualitzar les propietats del constructor pare. Aquestes propietats seran la definició dels controls, quins moviments pot fer i quins no, i com els ha de fer. Seguidament definirem l'esdeveniment del menú de pausa. Només cal definir aquest esdeveniment, ja que els anteriors ja estan inicialitzats quan fem l'*updateProperties.*

Finalment, eliminem les accions de la classe pare (ThirdPersonHandler), ja que les nostres accions seran diferents a les que ja hi han implementades, i les inicialitzem. Per inicialitzar-les, crearem un mètode nou: *inicialitzarAccions.*

```
actionDisparar = new DispararEMdPHandler( personatge );
addAction(actionDisparar, PROP_KEY_X, false);
```

Figura 76: Inicialització de l'acció de disparar

Per fer-ho, crearem una acció nova, passant-l'hi per paràmetre el que necessiti: en el

cas de l'acció de disparar paquets, com es veu a la Figura 76, simplement li passem per paràmetre un personatge, i, per afegir l'acció simplement cridarem al mètode *addAction*, de la classe **InputHandler**, al que l'hii passarem l'acció que volem executar, quin control l'executa (en el cas anterior, es veu que l'executa la tecla 'X'), i permet repeticions o no (en el cas de dispara no permet repeticions, però en els casos del moviment, endavant, endarrere, etc. si que permetem repeticions). A la resta d'accions el funcionament és igual: primer les creem i llavors les afegim.

```
public void update(float tpf){
      //Actualitzem la força que apliquem al personatge perqué es
//mantingui en peus
      personatge.update(tpf);
      //Actualitzem l'inputHandler pare, que es el thirdPersonHAndler,
      per //tal que ens giri el personatge
      super.update(tpf);
      //Activem l'animació d'esperar si no està moguent-se
      if(!this.nowTurning && !this.walkingForward && !
                  this.walkingBackwards ){
            if(!personatge.estaTirant()){
                  personatge.posaEsperant();
             }
      }
}
```

Figura 77: Mètode update de la classe EMdPHandler

Un cop inicialitzades les accions, ja estan llestes per controlar les entrades. Per tal de poder controlar-les necessitem un mètode anomenat *update* (com és pot veure a la Figura 77), que el que farà serà actualitzar el personatge. Per tal que a aquest li apliquin una força per mantenir-lo dempeus (com s'ha explicat anteriorment).

Seguidament actualitzarem el control d'entrada de la classe **ThirdPersonHandler**. Aquest serà el que executarà, en cas necessari, les accions ja inicialitzades. A més a més, també serà l'encarregat de fer girar el personatge de manera gradual, fent rotar la cara frontal del personatge.

Finalment, comprovem si el personatge es mou o dispara. En cas que no estigui fent res, executem l'animació de esperar.

8.7.2. Accions

Les accions, tal i com s'ha vist el el capítol de disseny, són classes que tenen dos mètodes importants; el constructor, que inicialitza els atributs necessaris, i el mètode

performAction que serà a on implementarem l'acció.

8.7.2.1. Accions de moviments

```
public void performAction(InputActionEvent event) {
      // Si el jugador estÃ  bloquejat, no fem res
      if(!((Personatge)handler.getJugador()).estaBloquejat(
                  Timer.getTimer().getTimeInSeconds())){
            handler.setGoingForward(true);
            Vector3f loc = handler.getTarget().getLocalTranslation();
            if (handler.isCameraAlignedMovement()) {
                rot.set(handler.getCamera().getDirection());
                rot.y = 0;
            } else {
               handler.getTarget().getLocalRotation().getRotationColumn(2,
                     rot);
      }
      rot.normalizeLocal();
      loc.addLocal(rot.multLocal((this.speed * event.getTime())));
      if(!personatge.estaCaminant())
      {
            personatge.posaCaminant();
      }

}
```

Figura 78: Mètode performAction

A la Figura 78 tenim el codi del mètode *performAction*, a on el que fem és, primer de tot, mirar si el jugador està bloquejat o no. Si està bloquejat no fem res. Altrament calculem cap a on mira el personatge, i sumem un valor en aquela direcció depenent de la velocitat. Finalment si l'animació de caminar no estava carregada, la carreguem. Això fem ja que, sinó, cada vegada l'acció tornaria a carregar-se des del principi i l'animació és reproduiria malament.

La resta d'accions de moviment són iguals, on les úniques línies que canvien són les que assignem quina acció estem duguen a terme amb el mètode, *setGoingForward,* i la del control de posició.

8.7.2.2. Acció de disparar

```
 GestorMusica.reproduirCrit(1,this.jugador.getWorldTranslation());
Paquet paquet = jugador.retornarEliminarPaquet();
this.jugador.getParent().attachChild(paquet);
logger.info("KeyNodeLlansarPaquet::El Jugador
"+this.jugador.getIdJugador()+" llanÃ§a un Paquet ");
paquet.getLocalTranslation().set(posJugador);
paquet.setHaEstatLlanÃ§at(true);
paquet.setActive(true);
paquet.clearDynamics();
Vector3f forsa = new Vector3f(direccioJugador);

forsa.set(forsa.mult(10).mult(this.jugador.getForÃ§a()));
paquet.addForce(forsa);
logger.info(forsa+"KeyNodeLlansarPaquet::Paquet llansat:
"+paquet.getNom());

((BLabel)GestorPantalla.getLlista().getLast().getFinestra().getComponent(
4*jugador.getIdJugador()+4)).setText(""+jugador.getNumPaquets());
if(!jugador.estaTirant())
{
      jugador.posaTirant();
}
jugador.actualitzarPaquets();
```

Figura 79: Codi que s'executa al disparar un paquet

L'acció de disparar, serà l'encarregada de llançar el paquet. El primer que es fa en aquesta acció és comprovar que el personatge no estigui bloquejat i que tingui paquets. Un cop feta aquesta comprovació, reproduïm el crit que farà el personatge al llançar-los, agafem el paquet del personatge, el col·loquem a la mà del jugador, i li apliquem una força segons cap a on estigui mirant aquest, tal i com es veu a la Figura 79.

Tamb actualitzem el número de paquets que té el jugador que és mostren per pantalla, i finalment activem l'animació de llançar paquets.

8.7.2.3. Acció del Menú de Pausa

L'acció del Menú de Pausa, serà una acció bastant senzilla, com es pot comprovar a la Figura 80, ja que simplement comprova que estigui la partida activa, i seguidament crida de la classe **GestorPantalla** el mètode p*ausarPartida* explicat anteriorment a l'apartat de Gestors.

```
public void performAction (InputActionEvent event) {
       if(GestorPantalla.esActivaPartida())
        {
              GestorPantalla.pausarPartida();
        }
}
```

Figura 80: Mètode que s'executa al pausar una partida

8.8. Implementació de models i optimització dels recursos

A l’hora de crear l’entorn i el personatge del joc, s’ha tingut en compte tots els factors de rendiment per a poder jugar al joc fluidament en els ordinadors disponibles actualment.

8.8.1. Número de polígons

Tots els models s’han fet seguint les guies del model Low-Poly. Aquest mètode et delimita el número de polígons que es poden fer servir en cada model, pensant en la importància del mateix, i si te animació (i quin tipus en cas de que en tingui), i la distancia de l’objecte respecte la càmera. Sempre s’ha de fer servir el menor número de polígons que permeti realitzar la feina amb la qualitat gràfica desitjada i consumint el mínim de recursos.

En el cas de l’escenari, com que treballem amb una càmera fixa, ha estat relativament fàcil determinar el número de polígons en cada model. A l’escenari hi ha un total de 5.949 polígons, més els personatges animats i les bústies. En l’actualitat, en un videojoc de guerra, un sol tanc pot tenir 10.000 polígons, de manera que creiem que la optimització en el nostre joc ha estat correcte.

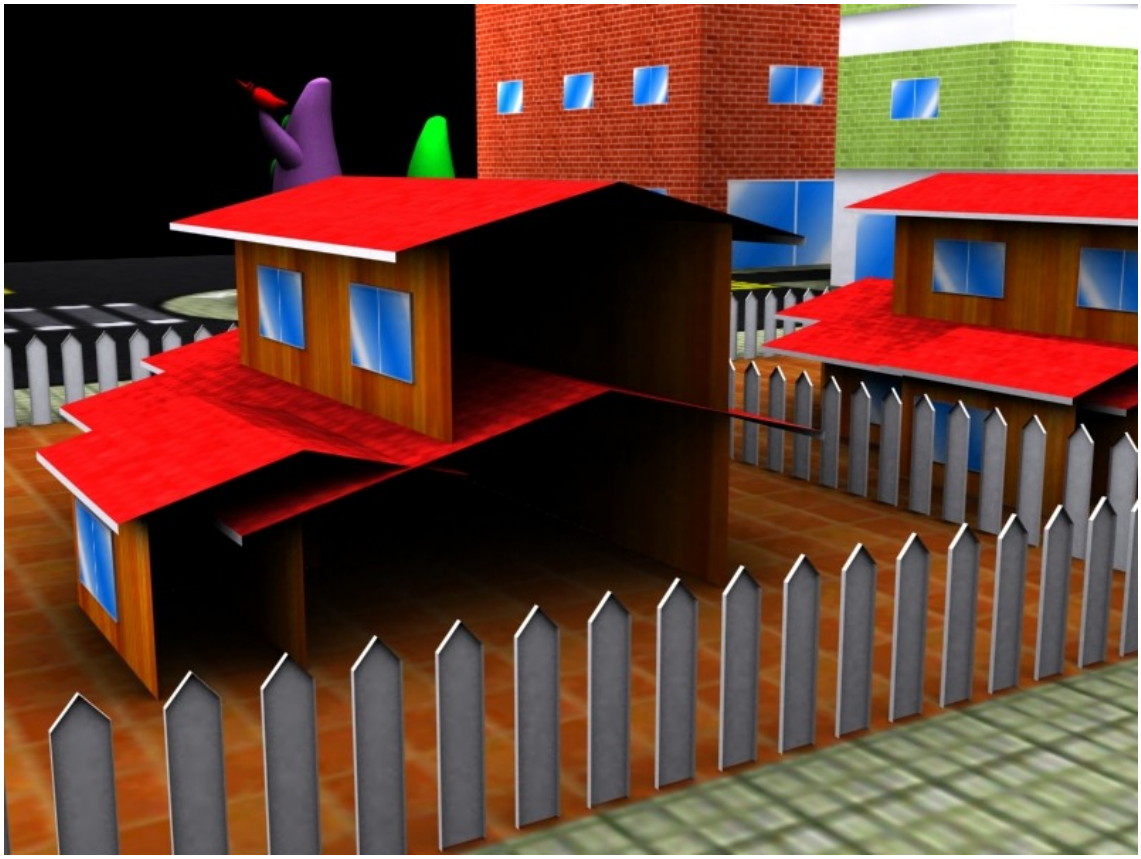


Figura 81: Mostra de les cases i tanques sense la part posterior

Per arribar a aquest número, i aprofitant les avantatges de la càmera fixa, s’han eliminat els polígons que queden fora de la vista de la càmera, com es veu a la Figura 81.

També a la Figura 81, podem veure que tant les cases com les tanques estan buides per darrera. D’aquesta manera estalviem polígons i textures en l’escenari, que repercutirà de forma positiva en el rendiment final del joc, ja que es una càrrega que evitem tant al processador com a la targeta gràfica i la memòria RAM amb informació innecessària.

8.8.2. Textures

Pel que fa les textures, es fan servir textures de 256x256 píxels, 512x512 píxels i 1024x1024 píxels. S’han fet servir sempre potencies de dos perquè es la manera en la que treballen els bancs de memòria del hardware gràfic i son mesures més fàcils de processar que les altres. Cada textura s’ha fet servir en un lloc concret tenint en compte els detalls i la distancia de la càmera en el que es troba l’objecte. Per

exemple, hi ha objectes més complexos, com les cases, que tenen textures més petites que el terra que es un simple pla però que ocupa més en pantalla, pel que ha de tenir una mida més gran. Els objectes com els edificis o la paret del fons, s’han hagut de reduir per evitar l’efecte d'*aliasing* que es produïa per l’excés de detalls en un espai massa reduït.

En les textures, també s’ha mirat de reduir el cost de procés de la CPU o la targeta gràfica, simulant les ombres que produirien les llums. D’aquesta manera, amb una llum ambiental que no provoca ombres (ni pròpies ni projectades), ja es pot tenir un bon resultat, com es veu a la Figura 82.

Figura 82: Detall de l'escenari amb textures ja ombrejades

Veiem unes ombres que han estat pre-renderitzades i enganxades a la textura per donar l’efecte d’una il·luminació amb la llum del sol.

Per altra banda, s’han afegit ombres per simular relleu i evitar-nos així més polígons. Com veiem en la Figura 83, les voreres dels carreres estan simulant en relleu amb una ombra en tot el seu contorn. D’aquesta manera simulem un relleu i la ombra

projectada per la llum del sol. En aquesta imatge apreciem també el que comentàvem anteriorment: les ombres difoses dels edificis, les cases i el text gravat al terra, també son enganxades a la textura del terra per simular la llum del sol, sense consumir recursos per calcular les mateixes.

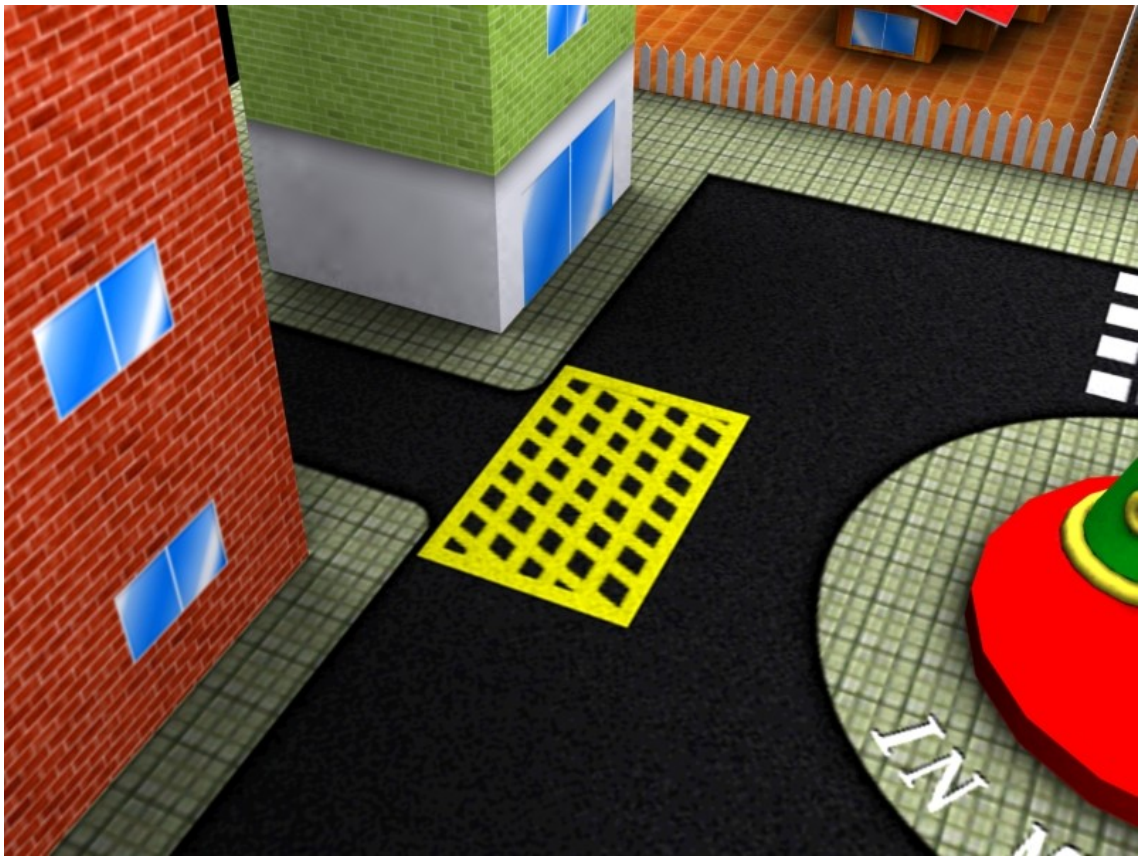


Figura 83: Imatge en detall de ombres simulant relleu.

9. Resultats

En aquest apartat comentarem els resultats obtinguts al final del desenvolupament.

9.1. Resultats de l'apartat comú

Figura 84: Caputra de pantalla de l'escenari carregat

A la Figura 84 podem veure un escenari carregat. Les propietats d'aquest escenari son la utilització de varis fitxers per al seva texturació, la il·luminació i col·locació de la càmera. A més, hi han els elements que conformen un escenari: les bústies, els personatges no jugadors i una furgoneta.

9.2. Resultats dels apartats d'en Pere Fonolleda

Aquí es mostraran captures de pantalla comentades de les parts que ha realitzat en Pere Fonolleda.

9.2.1. Animació per óssos

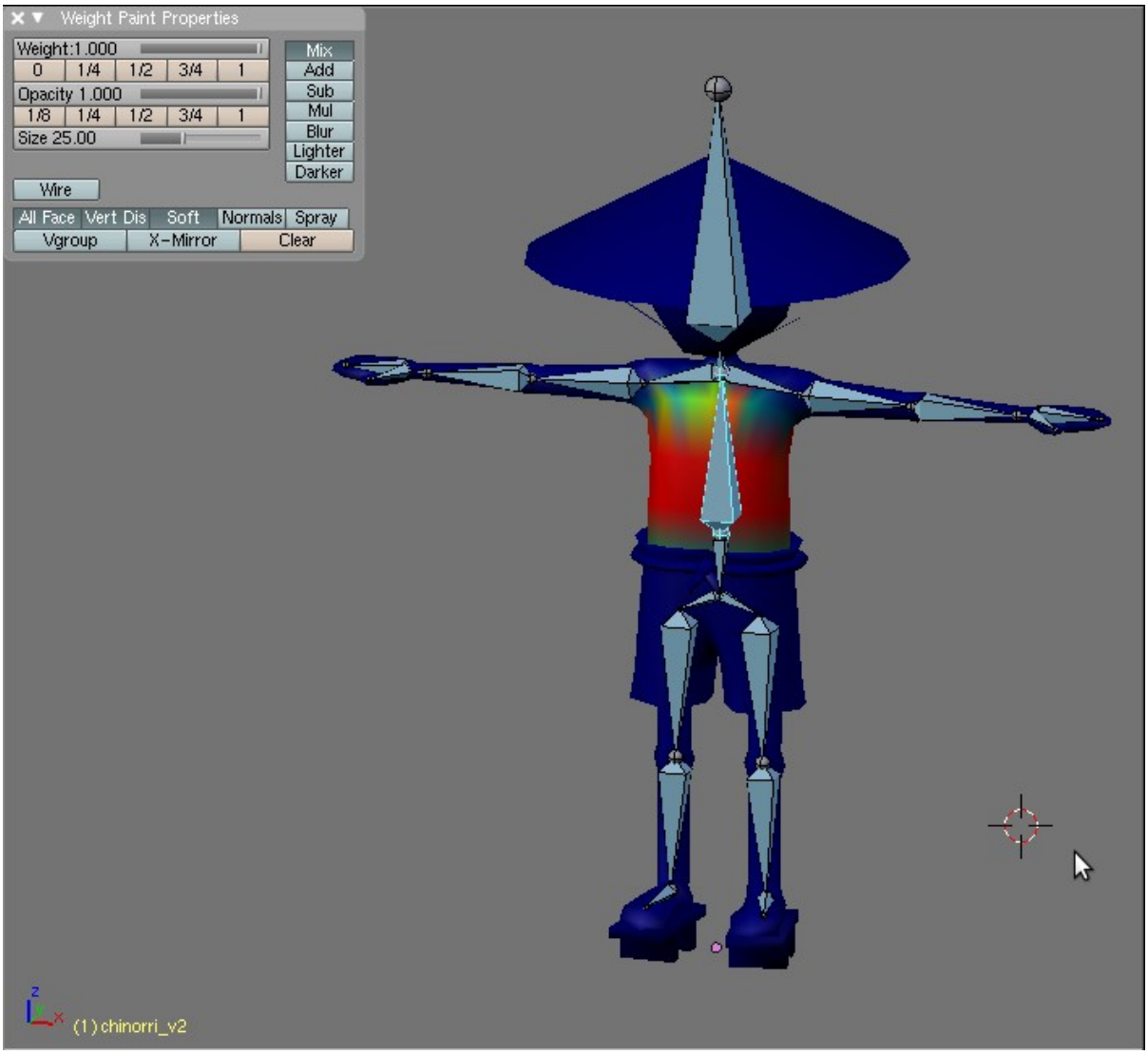


Figura 85: Unió del model amb l'esquelet

A la Figura 85 podem veure el procés de col·locació d'un esquelet a un personatge i l'adjudicació de parts de la malla als óssos per a la correcta deformació. Mitjançant les eines de pintura que es poden veure al requadre superior dret de la imatge, s'assiga un “pes” d'importància sobre l'os seleccionat a les malles que l'envolten. En la Figura 85 es pot veure com l'ós que recorre l'esquena afecta sobre tota la zona de l'abdomen, de color vermell, i no a la resta del cos, que segueix de color blau.

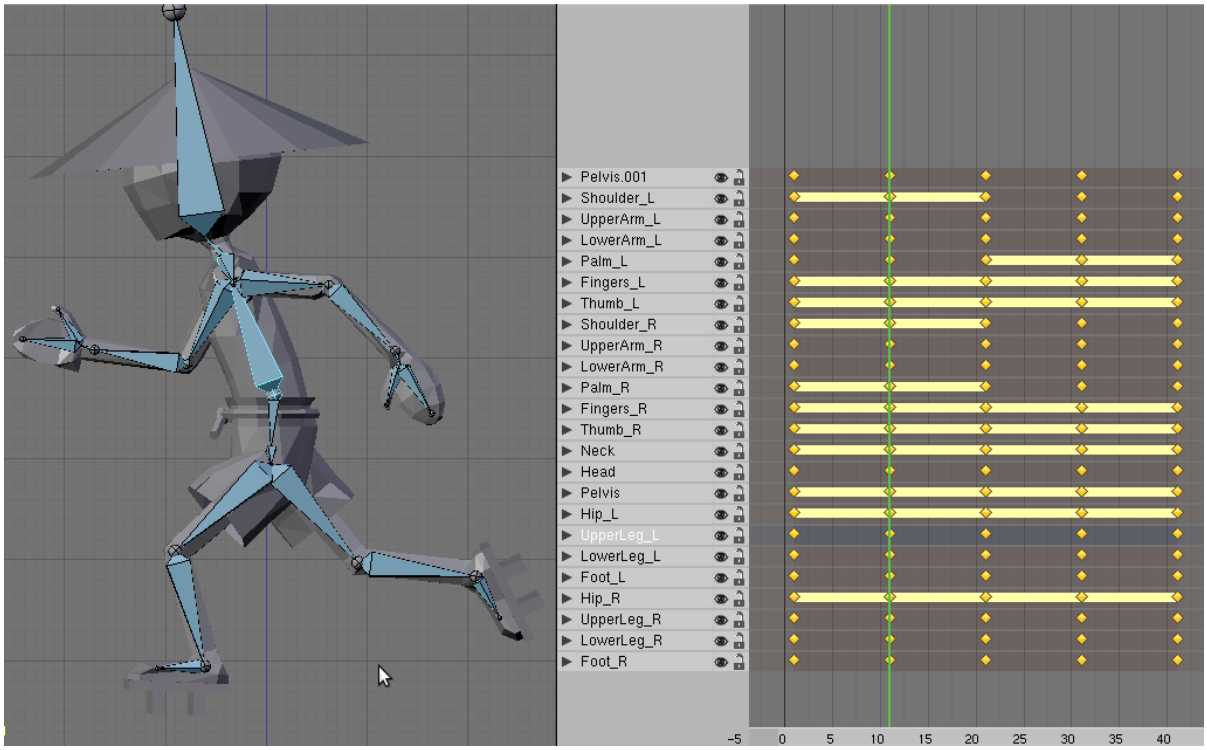


Figura 86: Animació del personatge amb el programa Blender

A la Figura 86 podem veure l'evolució del moviment dels óssos en un personatge en el programa en el que s'ha fet, el Blender. A la dreta de la pantalla tenim el model en moviment, amb els óssos visibles. Des d'aquí, per a cada *frame* de l'animació, podem col·locar els *bones* amb l'animació que volguem. A l'esquerra, tenim la línia de temps, on veiem com afecta a cada *bone* (noms a l'esquerra del diagrama) i els *frames* totals de l'animació.

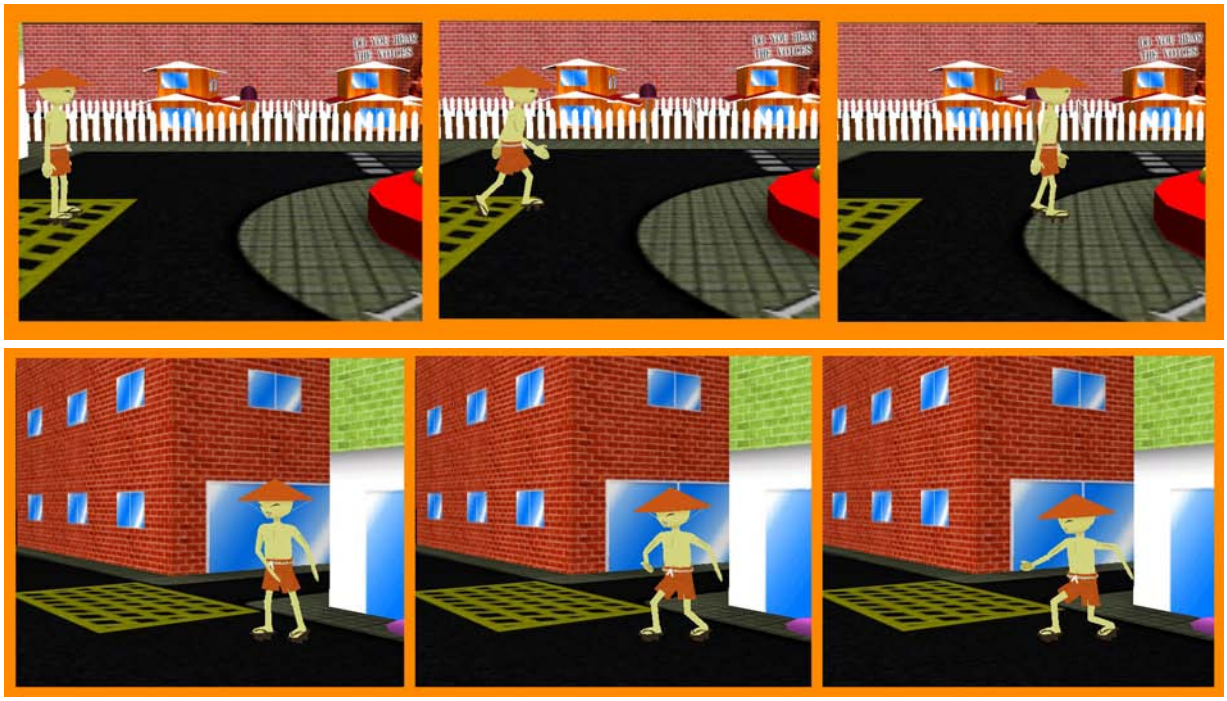


Figura 87: Models carregats i animats

A la Figura 87 podem veure el moviment d'aquest personatge un cop l'hem exportat dintre del nostre videojoc. La primera seqüència d'imatges mostra al personatge caminant, la segona fent el moviment de llençar un paquet.

9.2.2. Disseny de la interfície

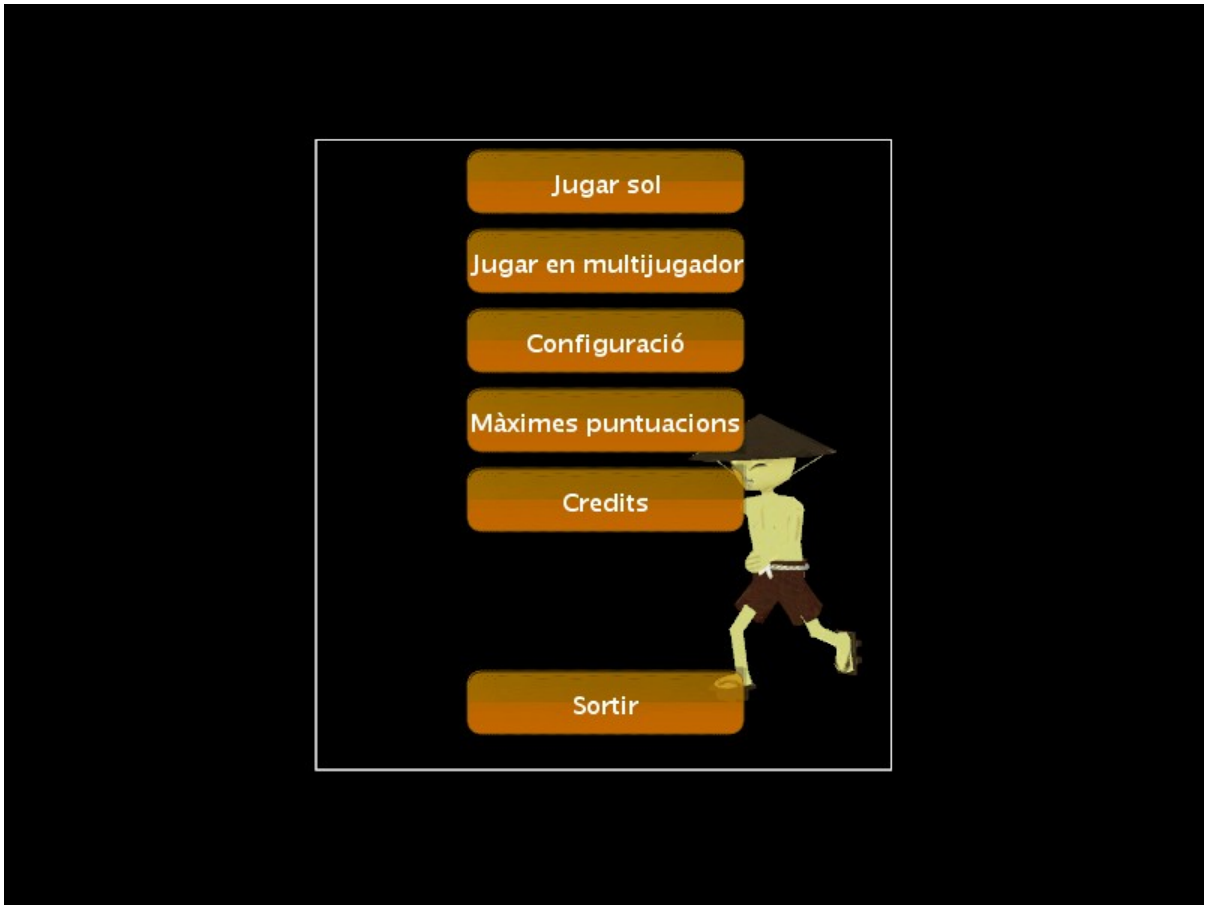


Figura 88: Menú principal.

A la Figura 88 podem veure la captura de pantalla del menú principal, amb un model carregat de fons fent el moviment de córrer.

Figura 89: Menú d'iniciar una partida en mode multijugador

A la Figura 89 veiem el menú d'iniciar partida amb un jugador. Hi podem veure els botons de “Torna” i “Juga!”, els selectors “Per temps” i “Per paquets”, on es veu que el botó “Per temps” està activat, i els selectors per fletxes i imatges per a seleccionar escenari i personatges, on cada jugador ha escollit un personatge diferent, i el jugador 4 no jugarà.

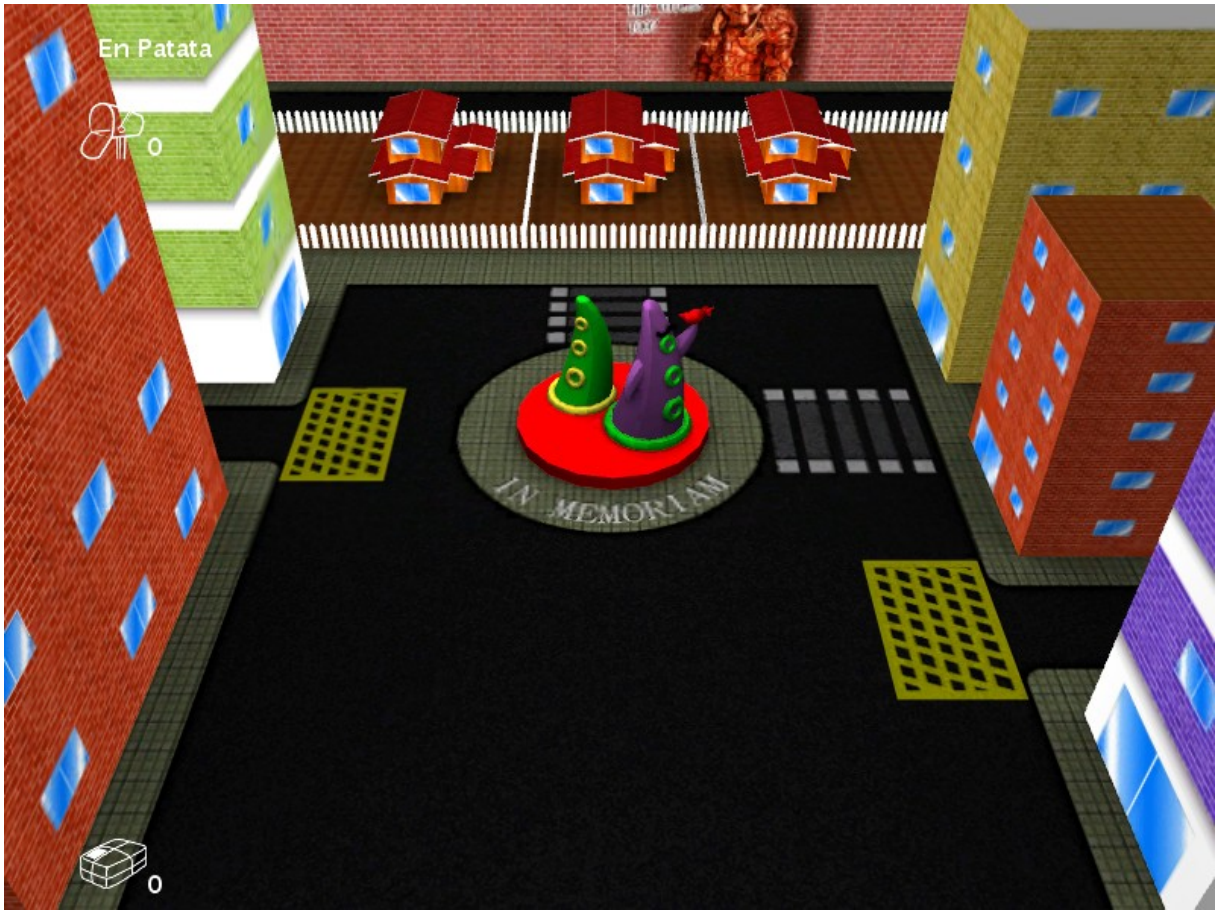


Figura 90: Imatge de la partida, on es veuen els icones indicadors

Finalment, a la Figura 90 es pot veure l'escenari carregat al iniciar una partida, i els icones creats a traves de la interfície gràfica per a mostrar el número de punts que té un jugador (la icona de la bústia), i el número de paquets que du a sobre (la icona del paquet).

9.2.3. Multijugador

Figura 91: Seqüència de joc en multijugador

A la Figura 91 podem veure una seqüència d'imatges de la partida creada amb tres personatges jugant, cada un d'un color diferent i movent-se per separat.

9.3. Resultats dels apartats d'en Ferran Font

Aquí es mostraran captures de pantalla comentades de les parts que ha realitzat en Ferran Font.

9.3.1. Disseny de la física

En aquest apartat observarem els resultats de la física del videojoc. Com a física s'entén totes aquelles col·lisions que es poden produir, i que s'han de tractar d'una manera especial.

La col·lisió més important és la del **Personatge** contra l'**Escenari,** és a dir, contra les parets que limiten l'àrea de joc, algun obstacle col·locat al centre, etc. Aquestes col·lisions simplement s'ha de controlar que el personatge no travessi aquests elements.Com es pot comprovar a la Figura 92, quan un personatge xoca contra la paret aquesta no el deixa avançar.

Figura 92: Seqüència que representen la col·lisió de un Personatge contra els elements de l'escenari.

En les altres col·lisions, controlarem quins son els elements que han xocat, executaran unes funcions o unes altres tal i com s'ha vist a l'apartat de Disseny, a la classe **ImplCallBack.**

9.3.2. Disseny de la Intel·ligència Artificial

Figura 93: Representació de com rebota un paquet per l'escenari.

Com hem explicat anteriorment, hi ha dos tipus de intel·ligència artificial, la dels personatges no jugadors, i la de les furgonetes. A la Figura 93 és pot veure com quan un personatge passa per davant d'un personatge no jugador, aquest el persegueix.

En el cas de la furgoneta, a la Figura 94 podem veure com actuarà en una partida, anant a carregar i a descarregar els paquets, segons els carregaments que porti.

Figura 94: Seqüencia d'entrada i sortida d'una furgoneta a l'escenari

9.3.3. Disseny de la Música

Els resultats obtinguts en aquest apartat no és poden mostrar en imatges, per tant els podrem observar el dia de la presentació mitjançant un petit vídeo dels efectes sonors del videojoc, així com la música dels menús, i durant la partida.

10. Conclusions

El nostre objectiu principal alhora de fer aquest projecte era la realització d'un videojoc senzill, des d'una idea inicial fins a un producte acabat. Podem concloure que aquest objectiu, com s'ha pogut comprovar, a estat assolit de forma satisfactòria, completant tots els requeriments definits al principi del projecte:

- Hem realitzat un videojoc complet, des d'un menú inicial fins al desenvolupament de la partida, tot encarat a l'usuari final.
- El nostre videojoc ha inclòs una escenificació 3D
- Càrrega de models amb animacions per óssos, texturat, i proves amb diferents formats de fitxers.
- Interacció entre elements gràcies a un motor de física.
- Implementació d'interfície gràfica.
- Implementació d'uns algoritmes d'intel·ligència artificial per a que la màquina controli alguns personatges.
- La possibilitat de jugar amb varis jugadors.
- La inclusió de música.

10.1. Conclusions personals

També podem afirmar que la nostra intenció, a nivell d'aprenentatge individual de conèixer una mica més a fons aquest món (el procés de creació d'un videojoc, tant el que es refereix a crear un projecte relacionat com al que es refereix a la programació encarada a aquest camp) ha estat satisfeta, ja que considerem que hem aprés moltes coses d'aquest camp que ens poden servir en un futur.

11. Treball futur

A partir de les bases assentades en aquest projecte, podríem fer un gran nombre de millores i modificacions al nostre videojoc.

Com a modificacions més destacables, enumerarem les següents:

- Ampliar el nombre de pantalles i jugadors. A més, com que el videojoc està preparat per a fer-ho dinàmicament, es podria fer una petita aplicació que comprovés que els nous models creats son correctes (mides estàndards, animacions implementades per als personatges, crear objectes que faran de bounding box en l'escenari, etc.).
- Una altra millora interessant seria la possibilitat de jugar per xarxa a aquest joc en el mode multijugador, fent una execució de client-servidor.
- Implementar algoritmes de *path-finding* per a que, apart de jugar contra altres jugadors humans en mode multijugador, es poguessin afegir *bots*, personatges que representarien jugadors però estarien controlats per la intel·ligència artificial.
- Afegir més animacions als personatges, de manera que al xocar, llençar un paquet, o simplement esperar, poguéssim carregar-les aleatòriament i donés més dinamisme al joc.
- Finalment, la possibilitat de no tant sols triar quin controlador es vol fer servir per jugar, sinó a més que poguessis configurar-hi els botons.

Apart dels punts aquí esmentats, tot allò relacionat amb el disseny que pugui ser millorat, però que ja no depèn dels programadors, sinó de l'ajuda d'un dissenyador extern.

12. Agraïments

En primer lloc, volem agrair al nostre tutor, en Gus, l'ajuda que hem rebut a l'hora de desenvolupar, a més de l'entusiasme mostrat, ja que el fet d'haver realitzat aquest projecte és, en bona mesura, “culpa” seva. També li volem agrair la paciència mostrada, i la implicació a les darreres fases del projecte per a poder-ho enllestir tot a temps.

En segon lloc, volem agrair al nostre amic Alberto els models que ha realitzat per l'aplicació (tots els models que hem fet servir han estat creats o adaptats per ell), i també la seva aportació de coneixements respecte a la creació de videojocs i la optimització, que ens han sigut de gran ajuda. A més, per les hores compartides de desconnexió amb el *DOW*.

En Guillem per la realització del tema musical principal del videojoc, i per futurs temes que ens ha promès que ens farà.

A la Yohana per donar-nos allotjament a on treballar junts, i per suportar les llargues nits de projecte reclosos.

A la Palmira per cuidar de nosaltres i donar-nos un cop de mà en la supervivència personal del tram final del projecte.

Als nostres pares per la paciència que han tingut amb nosaltres en una llarga carrera i suportar-nos, sobretot, en aquest darrer tram.

A els cunyats d'en Pere per el seu entusiasme en veure al xino caminar, fet que ens va motivar a accelerar el procés d'animació del personatge.

A tots els nostres amics i amigues que ens han aportat idees (més o menys vàlides...), suport moral, i cerveses fresques quan ja no podíem més.

Finalment, a la gent del fòrum del motor jMonkey Engine, que ens ha ajudat en procés de creació del videojoc, sigui responent als nostres dubtes o amb material del qual hem extret informació important.

A tots, moltes gràcies!

13. Bibliografia

Aquí farem un llistat amb tots aquells recursos importants que hem fet servir per al desenvolupament d'aquest projecte.

Users Guide. Wiki del jMonkey Engine.
http://www.jmonkeyengine.com/wiki/doku.php?id=user_s_guide

StandardGame, GameStates, and Multithreading (A New Way of Thinking). Wiki del jMonkey Engine.
http://jmonkeyengine.com/wiki/doku.php?id=standardgame_gamestates_and_multithreading_a_new_way_of_thinking

jME2 - Flag Rush Tutorial Series. JME 2.0 Tutorials Wiki.
http://www.jmonkeyengine.com/wiki/doku.php?id=jme2_-_flag_rush_tutorial_series

Creating your jME interface using GBUI. Pàgina del GBUI GoogleCode.
http://code.google.com/p/gbui/wiki/CreatingYourJmeInterfaceUsingGbui
Última modificació: 23/10/2008

HottBJ, Handy Object Transfer Tool, Blender to jME. JME 2.0 Tutorials Wiki.
http://www.jmonkeyengine.com/wiki/doku.php?id=blenderjme_basics_tutorial
Última modificació: 27/08/2009 per en blaine.

Tutorials. jME Physics 2 Wiki.
http://wiki.jmephysics.irrisor.net/tiki-index.php?page=Tutorials

Tutorials/Animation/BSoD/Character Animation/Setting up the mesh. Blender Wiki.
http://wiki.blender.org/index.php/Doc:Tutorials/Animation/BSoD/Character_Animation/Setting_up_the_mesh
Última modificació: 04/04/2009